Pattern matching with don't cares and few errors

Raphaël Clifford, Klim Efremenko, Ely Porat, Amir Rothschild

Journal of Computer and System Sciences **76** (2010) 115–124.

Speaker: Joseph Chuang-Chieh Lin

Genomics Research Center, Academia Sinica
Taiwan

23 May 2014

Pattern matching

Σ: alphabet

$t = t_1 t_2 \ldots t_n \in \Sigma^n$: text

$p = p_1 p_2 \ldots p_m \in \Sigma^m$: pattern

Exact pattern matching

Given: a pattern $p \in \Sigma^n$, a text $t \in \Sigma^m$.

Goal: find all the places that p matches t.

- $O(n)$ time [Boyer & Moore 1977; Knuth, Morris & Pratt 1977]

Pattern matching

Pattern matching with don't cares

Given: a pattern $p \in \Sigma^n$, a text $t \in \Sigma^m$, where p, t may contain 'ϕ's.
Goal: find all the places that p matches t.

- $\Theta(n \log m)$ [Cole& Hariharan 2002; Clifford$\times$2 & 2007].

Pattern matching (k-mismatch)

k-mismatch WITHOUT don't cares

Given: a pattern $p \in \Sigma^n$, a text $t \in \Sigma^m$ and an integer $k \geq 0$.
Goal: find all the places that p matches t with $\leq k$ mismatches.

- $\Theta(n\sqrt{m \log m})$ time [Abrahamson 1987; Kosaraju 1987].
- $\Theta(n\sqrt{k \log k})$ [Amir, Lewenstein & Porat 2004].

Pattern matching (k-mismatch)

k-mismatch with don't cares

Given: a pattern $p \in \Sigma^n$, a text $t \in \Sigma^m$ and an integer $k \geq 0$, where p, t may contain 'ϕ's.
Goal: find all the places that p matches t with $\leq k$ mismatches.

- $O(nm^{1/3}k^{1/3}\log^{2/3} m)$ time [Clifford & Porat 2007].
  - 'ϕ's: permitted in either the pattern or text, but not both.
- Extend Kosaraju & Abrahamson's work with little extra work: $\Theta(n\sqrt{m \log m})$ time.
- No other previous efficient algorithm for this problem.

Contribution of this paper

k-mismatch with don't cares

Given: a pattern $p \in \Sigma^n$, a text $t \in \Sigma^m$ and an integer $k \geq 0$, where p, t may contain 'ϕ's.

Goal: find all the places that p matches t with $\leq k$ mismatches.

This paper:

- Two randomized $\tilde{\Theta}(nk)$ time algorithms.
  - A randomized $\Theta(nk \log m \log n)$ time algorithm.
  - Further improved $\rightarrow \Theta(n(k + \log m \log k) \log n)$ time.
- A deterministic $\Theta(nk^2 \log^2 m)$ time algorithm (group testing).
- $\Theta(nk \operatorname{polylog} m)$ time using k-selectors.

Contribution of this paper

k-mismatch with don't cares

Given: a pattern $p \in \Sigma^n$, a text $t \in \Sigma^m$ and an integer $k \geq 0$, where p, t may contain 'ϕ's.
Goal: find all the places that p matches t with $\leq k$ mismatches.

This paper:

- Two randomized $\tilde{\Theta}(nk)$ time algorithms.
  - A randomized $\Theta(nk \log m \log n)$ time algorithm.
  - Further improved $\rightarrow \Theta(n(k + \log m \log k) \log n)$ time.
- A deterministic $\Theta(nk^2 \log^2 m)$ time algorithm (group testing).
- $\Theta(nk \operatorname{polylog} m)$ time using k-selectors.

k-mismatch & Hamming distance

- $HD(i)$: the Hamming distance between p and $t[i, \ldots, i+m-1]$.
  - $\star$ ϕ matches any symbol in Σ.
- $$HD_k(i) = \begin{cases} HD(i) & \text{if } HD(i) \leq k \\ \perp & \text{otherwise.} \end{cases}$$
- $HD_k(i) \neq \perp$
  $\Rightarrow$ There is a k-mismatch between p and t at alignment i.
- For example, $HD_2(1) = 2$, $HD_2(5) = \perp$

i: 1 2 3 4 5 6 7 8 9 10
t: A A C ϕ G A ϕ T T G
p: A ϕ G G A

k-mismatch & Hamming distance

- $HD(i)$: the Hamming distance between p and $t[i,\ldots,i+m-1]$.
  - $\star$ ϕ matches any symbol in Σ.
- $$HD_k(i) = \begin{cases} HD(i) & \text{if } HD(i) \leq k \\ \perp & \text{otherwise.} \end{cases}$$
- $HD_k(i) \neq \perp$
  $\Rightarrow$ There is a k-mismatch between p and t at alignment i.
- For example, $HD_2(1) = 2$, $HD_2(5) = \perp$.

  i: 1 2 3 4 5 6 7 8 9 10

  t: A A C ϕ G A ϕ T T G

  p: A ϕ G G A

k-mismatch & Hamming distance

- $HD(i)$: the Hamming distance between p and $t[i, \ldots, i+m-1]$.
  - ⋆ ϕ matches any symbol in Σ.
-
$$HD_k(i) = \begin{cases} HD(i) & \text{if } HD(i) \leq k \\ \perp & \text{otherwise.} \end{cases}$$
- $HD_k(i) \neq \perp$
  $\Rightarrow$ There is a k-mismatch between p and t at alignment i.
- For example, $HD_2(1) = 2$, $HD_2(5) = \perp$.

  i: 1 2 3 4 5 6 7 8 9 10

  t: A A C ϕ G A ϕ T T G

  p: A ϕ G G A

k-mismatch & Hamming distance

- $HD(i)$: the Hamming distance between p and $t[i, \ldots, i+m-1]$.
  - ϕ matches any symbol in Σ.
- $$HD_k(i) = \begin{cases} HD(i) & \text{if } HD(i) \leq k \\ \perp & \text{otherwise.} \end{cases}$$
- $HD_k(i) \neq \perp$
  $\Rightarrow$ There is a k-mismatch between p and t at alignment i.
- For example, $HD_2(1) = 2$, $HD_2(5) = \perp$.

  i: 1 2 3 4 5 6 7 8 9 10

  t: A A C ϕ G A ϕ T T G

  p: A ϕ G G A

Transformation from matching to coding...

- The key observation by [Clifford×2 2007]:

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

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

where

$$p'_j = \begin{cases} 0 & \text{if } p_j = `\phi' \\ 1 & \text{otherwise;} \end{cases} \quad t'_i = \begin{cases} 0 & \text{if } t_i = `\phi' \\ 1 & \text{otherwise.} \end{cases}$$

- The sum equals 0 iff there is an exact match with don't cares.

Transformation from matching to coding...

- The key observation by [Clifford×2 2007]:

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

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

where

$$p'_j = \begin{cases} 0 & \text{if } p_j = \text{`}\phi\text{'} \\ 1 & \text{otherwise;} \end{cases} \quad t'_i = \begin{cases} 0 & \text{if } t_i = \text{`}\phi\text{'} \\ 1 & \text{otherwise.} \end{cases}$$

- The sum equals 0 iff there is an exact match with don't cares.

The cross-correlation

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

- The above cross-correlation (convolution) can be calculated in $\Theta(n \log m)$ time.
  - Fast Fourier Transform (FFT).

Starting by 1-mismatch...

The 1-mismatch problem

To determine where p and t have EXACTLY ONE mismatch.

- Masking out a number of positions in p with ϕ's at random (each of prob. $(k-1)/k$).
  - The resulting pattern: *subpattern*.
  - ⋆ It is likely that exactly one mismatch can be found.

Starting by 1-mismatch...

The 1-mismatch problem

To determine where p and t have EXACTLY ONE mismatch.

- Masking out a number of positions in p with ϕ's at random (each of prob. $(k-1)/k$).
  - The resulting pattern: *subpattern*.
  - ⋆ It is likely that exactly one mismatch can be found.

Algorithm 1-mismatch $\Theta(n \log m)$

Input: Pattern p, text t.
Output: $B[i]$: mismatch location in t for each aligment where $HD(i) = 1$.

1. Compute $A_0[i] = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
2. Compute $A_1[i] = \sum_j (i+j-1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
3. for each $i \in \{0, 1, \ldots, n\}$ do
   a. if $A_0[i] \neq 0$ then
      - $B[i] \leftarrow A_1[i]/A_0[i]$;

   b. else
      - $B[i] \leftarrow$ No_Mismatch;
4. for each $i \in \{0, 1, \ldots, n\}$ s.t. $B[i] \neq \texttt{No\_Mismatch}$ do
   - if $(p[B[i] - i + 1] - t[B[i]])^2 \neq A_0[i]$ then $B[i] \leftarrow \texttt{More\_Than\_1\_Mismatch}$;

Algorithm 1-mismatch $\Theta(n \log m)$

Input: Pattern p, text t.
Output: $B[i]$: mismatch location in t for each aligment where $HD(i) = 1$.

1. Compute $A_0[i] = \sum_j (p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
2. Compute $A_1[i] = \sum_j (i + j - 1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}$;
3. for each $i \in \{0, 1, \ldots, n\}$ do
   a. if $A_0[i] \neq 0$ then
      - $B[i] \leftarrow A_1[i]/A_0[i]$; $\leftarrow$ DECODING

   b. else
      - $B[i] \leftarrow$ No_Mismatch;
4. for each $i \in \{0, 1, \ldots, n\}$ s.t. $B[i] \neq$ `No_Mismatch` do
   - if $(p[B[i] - i + 1] - t[B[i]])^2 \neq A_0[i]$ then $B[i] \leftarrow$ `More_Than_1_Mismatch`;

The overall strategy of the first randomized algorithm

- Running `1-mismatch` for $\Theta(k \log n)$ times.
- Each time we get the location of a mismatch (with the subpattern) if one occurs.
- Similar to the concept of solving the Coupon Collector's problem.
- Yet, *the last coupon* is always the most difficult to get!

The overall strategy of the first randomized algorithm

- Running `1-mismatch` for $\Theta(k \log n)$ times.
- Each time we get the location of a mismatch (with the subpattern) if one occurs.
- Similar to the concept of solving the Coupon Collector's problem.
- Yet, *the last coupon* is always the most difficult to get!

The overall strategy of the first randomized algorithm

- Running `1-mismatch` for $\Theta(k \log n)$ times.
- Each time we get the location of a mismatch (with the subpattern) if one occurs.
- Similar to the concept of solving the Coupon Collector's problem.
- Yet, *the last coupon* is always the most difficult to get!

The overall strategy of the first randomized algorithm

- Running `1-mismatch` for $\Theta(k \log n)$ times.
- Each time we get the location of a mismatch (with the subpattern) if one occurs.
- Similar to the concept of solving the Coupon Collector's problem.
- Yet, *the last coupon* is always the most difficult to get!

Time complexity: $\Theta(nk \log n \log m)$.

The overall strategy of the first randomized algorithm

- Running `1-mismatch` for $\Theta(k \log n)$ times.
- Each time we get the location of a mismatch (with the subpattern) if one occurs.
- Similar to the concept of solving the Coupon Collector's problem.
- Yet, *the last coupon* is always the most difficult to get!

Time complexity: $\Theta(nk \log n \log m)$.

Correctness?

Correctness of the first randomized algorithm

- For $HD(i) \leq k$:
  - One fixed single mismatch is found in one iteration with prob. $\geq \left(\frac{k-1}{k}\right)^{k-1} \cdot \frac{1}{k} > \frac{1}{ek}$.
    - This one is not found after $\Theta(k \log n)$ iterations with prob. $\leq (1 - 1/ek)^{\Theta(k \log n)} \leq n^{-c}$ for some constant c.
  - Applying union bound to derive the overall error prob.
- For $HD(i) > k$:
  - Extra checking stage by computing

$$C[i] = \sum_{j=1}^{m} (i + j - 1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}, \text{ for each } i.$$

  - "Correct" $C[i]$ for each distinct 1-mismatch using the found $A_0[i]$.
    - Keep track of found mismatches (using a binary search tree).

Correctness of the first randomized algorithm

- For $HD(i) \leq k$:
  - One fixed single mismatch is found in one iteration with prob. $\geq \left(\frac{k-1}{k}\right)^{k-1} \cdot \frac{1}{k} > \frac{1}{ek}$.
    - This one is not found after $\Theta(k \log n)$ iterations with prob. $\leq (1 - 1/ek)^{\Theta(k \log n)} \leq n^{-c}$ for some constant c.
  - Applying union bound to derive the overall error prob.
- For $HD(i) > k$:
  - Extra checking stage by computing

$$C[i] = \sum_{j=1}^{m} (i+j-1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}, \text{ for each } i.$$

  - "Correct" $C[i]$ for each distinct 1-mismatch using the found $A_0[i]$.
    - Keep track of found mismatches (using a binary search tree).

Correctness of the first randomized algorithm

- For $HD(i) \leq k$:
  - One fixed single mismatch is found in one iteration with prob. $\geq \left(\frac{k-1}{k}\right)^{k-1} \cdot \frac{1}{k} > \frac{1}{ek}$.
    - This one is not found after $\Theta(k \log n)$ iterations with prob. $\leq (1 - 1/ek)^{\Theta(k \log n)} \leq n^{-c}$ for some constant c.
  - Applying union bound to derive the overall error prob.
- For $HD(i) > k$:
  - Extra checking stage by computing

$$C[i] = \sum_{j=1}^{m} (i + j - 1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}, \text{ for each } i.$$

  - "Correct" $C[i]$ for each distinct 1-mismatch using the found $A_0[i]$.
    - Keep track of found mismatches (using a binary search tree).

Correctness of the first randomized algorithm

- For $HD(i) \leq k$:
  - One fixed single mismatch is found in one iteration with prob. $\geq \left(\frac{k-1}{k}\right)^{k-1} \cdot \frac{1}{k} > \frac{1}{ek}$.
    - This one is not found after $\Theta(k \log n)$ iterations with prob. $\leq (1 - 1/ek)^{\Theta(k \log n)} \leq n^{-c}$ for some constant c.
  - Applying union bound to derive the overall error prob.
- For $HD(i) > k$:
  - Extra checking stage by computing

$$C[i] = \sum_{j=1}^{m} (i + j - 1)(p_j - t_{i+j-1})^2 p'_j t'_{i+j-1}, \text{ for each } i.$$

  - "Correct" $C[i]$ for each distinct 1-mismatch using the found $A_0[i]$.
    - Keep track of found mismatches (using a binary search tree).

The clever ideas

- Instead of finding *ALL* mismatches at once, find HALF of them!
  - Efficient & with high prob. of success.
- Save the effort in dealing with previous found mismatches.
  - Correct the sums in A_0 and A_1 prior to their being used.
- The left mismatches to be found $\longrightarrow$ increase the sampling rate!

The clever ideas

- Instead of finding *ALL* mismatches at once, find HALF of them!
  - Efficient & with high prob. of success.
- Save the effort in dealing with previous found mismatches.
  - Correct the sums in A_0 and A_1 prior to their being used.
- The left mismatches to be found $\longrightarrow$ increase the sampling rate!

The clever ideas

- Instead of finding *ALL* mismatches at once, find HALF of them!
  - Efficient & with high prob. of success.
- Save the effort in dealing with previous found mismatches.
  - Correct the sums in A_0 and A_1 prior to their being used.
- The left mismatches to be found $\rightarrow$ increase the sampling rate!

A faster recursive randomized algorithm

Input: Pattern p, text t, and an integer $k \geq 0$.
Output: Array $O[i] = HD_k(p, t[i, \ldots, i + m - 1])$.

1. Initialize E and set $k_0 \leftarrow k$;
2. for $s \leftarrow 0$ to $\lfloor \log k \rfloor$ do
   a. for times $\leftarrow 1$ to $\Theta(k_s + \log n)$ do /* Sample and Match stage */
      I. Sample subpattern p^* with sample rate $1/k_s$;
      II. `self-correcting-1-mismatch`(p^*, t, E);
   b. Update E according to the mismatches found in the iterations;
   c. $k_{s+1} \leftarrow k_s/2$;
3. $L[i] \leftarrow$ total number of distinct mismatches found at alignment i;
4. Check at each position i in t that all mismatches were found;
5. $O[i] \leftarrow L[i]$, if all mismatches were found, otherwise $O[i] \leftarrow \perp$.

A faster recursive randomized algorithm

Input: Pattern p, text t, and an integer $k \geq 0$.
Output: Array $O[i] = HD_k(p, t[i, \ldots, i + m - 1])$.

1. Initialize E and set $k_0 \leftarrow k$;
2. for $s \leftarrow 0$ to $\lfloor \log k \rfloor$ do
   a. for times $\leftarrow 1$ to $\Theta(k_s + \log n)$ do /* Sample and Match stage */
      I. Sample subpattern p^* with sample rate $1/k_s$;
      II. `self`
   b. Update E according to the mismatches found in the iterations;
   c. $k_{s+1} \leftarrow k_s/2$;
3. $L[i] \leftarrow$ total number of distinct mismatches found at alignment i;
4. Check at each position i in t that all mismatches were found;
5. $O[i] \leftarrow L[i]$, if all mismatches were found, otherwise $O[i] \leftarrow \perp$.

$O(nk/k_s)$ times that a previously discovered mismatch is found whp.

A faster recursive randomized algorithm

Input: Pattern p, text t, and an integer $k \geq 0$.
Output: Array $O[i] = HD_k(p, t[i, \ldots, i + m - 1])$.

1. Initialize E and set $k_0 \leftarrow k$;
2. for $s \leftarrow 0$ to $\lfloor \log k \rfloor$ do
   a. for times $\leftarrow 1$ to $\Theta(k_s + \log n)$ do /* Sample and Match stage */
      I. Sample subpattern p^* with sample rate $1/k_s$;
      II. self
   b. Update E accordin
   c. $k_{s+1} \leftarrow k_s/2$;
3. $L[i] \leftarrow$ total number of distinct mismatches found at alignment i;
4. Check at each position i in t that all mismatches were found;
5. $O[i] \leftarrow L[i]$, if all mismatches were found, otherwise $O[i] \leftarrow \perp$.

$O(nk/k_s)$ times that a previously discovered mismatch is found whp.

$\leq k_s/2$ different mismatches will remain to be found whp.

A faster recursive randomized algorithm $\Theta(n(k + \log m \log k) \log n)$

Input: Pattern p, text t, and an integer $k \geq 0$.
Output: Array $O[i] = HD_k(p, t[i, \ldots, i + m - 1])$.

1. Initialize E and set $k_0 \leftarrow k$;
2. for $s \leftarrow 0$ to $\lfloor \log k \rfloor$ do
   a. for times $\leftarrow 1$ to $\Theta(k_s + \log n)$ do /* Sample and Match stage */
      I. Sample subpattern p^* with sample rate $1/k_s$;
      II. self

      $O(nk/k_s)$ times that a previously discovered mismatch is found whp.

   b. Update E accordin
   c. $k_{s+1} \leftarrow k_s/2$;

   $\leq k_s/2$ different mismatches will remain to be found whp.

3. $L[i] \leftarrow$ total number of distinct mismatches found at alignment i;
4. Check at each position i in t that all mismatches were found;
5. $O[i] \leftarrow L[i]$, if all mismatches were found, otherwise $O[i] \leftarrow \perp$.

Thanks for your attention.

Appendix

Half different mismatches can be found whp in each stage

- $k_s = k/2^s$

Lemma 4.5

After $\Theta(k_s + \log n)$ iterations of the sample and match stage, the locations and values of $\leq k_s/2$ different mismatches will remain to be found whp.

- X_i: 1 if a mismatch at the ith stage is found, 0 otherwise.
- $E(\sum_{i=1}^{\omega} X_i) \geq \omega/e$, for $\omega = \Theta(k_s + \log n)$.
  - ⋆ Sum of random variables → using Chernoff bounds.
  - $\geq \omega/2e$ mismatches can be found whp.
- Found mismatches are entirely contained in any size-$(k_s/2)$ set with prob. $\leq 2^{\omega/2e}\binom{k_s}{k_s/2}$.

The number of times to handling previously found mismatches

Lemma 4.7

If the sample and match stage is run $r = \Theta(k_s + \log n)$ times, # times a previously discovered mismatch is found is $O(nk(k_s + \log n)/k_s)$ whp.

- $X_{i,j}$: whether p_i was replaced with ϕ in iteration j.
  - $\Pr(X_{i,j} = 1) = 1/k_s$.
- a_i: # mismatches previously found at p_i in all alignments.
- $X = \sum_{i \in [m], j \in [r]} a_i X_{i,j}$ is $O(nk(k_s + \log n)/k_s)$ whp.
  - Using Chernoff bound once again (though a different formula).

⋆ The overall time complexity can be derived easily.

The Chernoff-Hoeffding bounds

Theorem 4.4

Assume that $X_1, \ldots, X_m$ are i.i.d. random variables, $X_i \in \{0,1\}$. Let $\mu_i = E(X_i)$. Then

$$Pr\left(\sum_{i=1}^{m} X_i \leq (1-\delta)m\mu\right) < e^{-m\mu\delta^2/2}.$$

Theorem 4.6

Let $X_1, \ldots, X_m$ be discrete, independent random variables s.t. $E(X_i) = 0$ and $|X_i| \leq 1$ for all i. Let $X = \sum_{i=1}^{m} X_i$. Then

$$Pr(X \geq \lambda\sqrt{\mathrm{Var}(X)}) \leq e^{-\lambda^2/4}.$$

In the proof of Theorem 4.9

Let us concentrate on the ith stage of the recursion. At this stage, we need to solve the $k/2^i$-mismatch problem, by running the self-correcting 1-mismatch algorithm $\Theta(2^i + \log n)$ times. ...

- Shouldn't it be $\Theta(k/2^i + \log n)$ times?
    - Though it doesn't affect the result.