Approximating Longest Common Substring with k mismatches

Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Tatiana Starikovskaya

Similarity measures

Given two strings X and Y, how similar are they?

Ideally, we want a similarity measure that is

- Robust: Small change in the input $\Rightarrow$ small change of the measure
- Fast to compute

Applications in **Bioinformatics**, **Information Retrieval**.

Edit distance

Smallest number of **insertions**, **deletions**, and **substitutions** required to convert one string into the other.

$$\text{EditDistance}(\text{GATTACAT}, \text{ATTACATT}) = 2$$

Can be computed in quadratic time using dynamic programming. This is probably optimal:

[Backurs and Indyk'15] The Edit distance can't be computed in strongly subquadratic time, unless SETH is false.

SETH (Strong Exponential Time Hypothesis)
$\forall \delta > 0$, there exists an integer q such that SAT on q-CNF formulas with m clauses and n variables cannot be solved in time $m^{O(1)} 2^{(1-\delta)n}$.

Longest Common Substring

The maximal length of a string that occurs in both strings.

LCS (T**AAG**C, **AAG**AA) = 3

Can be computed in $\mathcal{O}(n)$ time **[Hui'92]**.

Unfortunately, not robust: can change a lot when we change a few characters of the input.

This work

Longest Common Substring with k mismatches problem

Input: an integer $\mathbf{k}$, strings $\mathbf{S_1}, \mathbf{S_2}$ of length $\mathbf{n}$

Output: the maximal length of a substring of $\mathbf{S_1}$ that occurs in $\mathbf{S_2}$ with $\mathbf{k}$ mismatches

$$\mathrm{LCS_k}\ (\mathrm{T}\mathbf{AAGC},\ \mathbf{AAGA}\mathrm{A}) = 4 \text{ for } k = 1$$

Closely related to the k-macs (the k-mismatch average common substring) distance **[Leimeister, Morgenstern'14]**

Longest Common Substring with k mismatches

Exact solutions:

- $k = 1$: $\mathcal{O}(n \log n)$ time **[Flouri et al.'15]**
- $\mathcal{O}(n^2)$ time - dyn. prog. **[Flouri et al.'15]**
- $\mathcal{O}(n((k+1)(|\mathrm{LCS}|+1))^k)$ or $\mathcal{O}(n^2|\mathrm{LCS_k}|/k)$ time **[Grabowski'15]**
- $k^{1.5}n^2/2^{\Omega(\sqrt{\frac{\log n}{k}})}$ time, rand. **[Abboud et al.'15]**
- $\mathcal{O}(n \log^k n)$ time **[Thankachan et al.'16]**
- $\mathrm{LCS_k} \geq \log^{2k+2} n$: $\mathcal{O}(n)$ time **[Charalampopoulos et al.'18]**

All solutions use $\mathcal{O}(n)$ space.

In general, LCS_k cannot be solved in strongly subquadratic time, unless SETH is false **[Kociumaka et al.'19]**

Longest Common Substring with approx. k mismatches

Input: an integer $\mathbf{k}$, a constant $\varepsilon > \mathbf{0}$, strings $\mathbf{S_1}, \mathbf{S_2}$ of length $\mathbf{n}$

Output: The length $\mathrm{LCS}_{\tilde{k}} \geq \mathrm{LCS}_k(\mathbf{T_1}, \mathbf{T_2})$ of a substring of $\mathbf{S_1}$ that occurs in $\mathbf{S_2}$ with $\leq (\mathbf{1} + \varepsilon) \cdot \mathbf{k}$ mismatches

$$S_1 = T\mathbf{AAGCTT}T,\ S_2 = C\mathbf{ACGTTT}C,\ k = 2,\ \varepsilon = 1.5$$
$$\mathrm{LCS}_k(S_1, S_2) = 6 \Rightarrow \text{we can return } \mathbf{AGCTTT}$$

- More robust than LCS, easier to compute
- $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time, $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ space for any $0 < \varepsilon < 2$ **[Kociumaka et al.'19]**
- Main idea: locality-sensitive hashing
- Very complex system of hash functions, superlinear space

Longest Common Substring with approx. k mismatches

Input: an integer $\mathbf{k}$, a constant $\varepsilon > \mathbf{0}$, strings $\mathbf{S_1}, \mathbf{S_2}$ of length $\mathbf{n}$

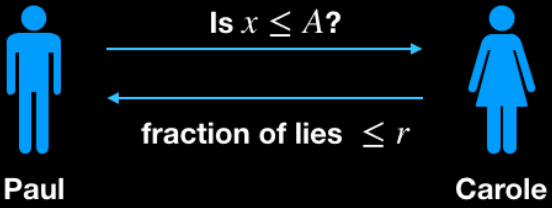
Output: The length $\mathrm{LCS}_{\tilde{k}} \geq \mathrm{LCS}_k(\mathbf{T_1}, \mathbf{T_2})$ of a substring of $\mathbf{S_1}$ that occurs in $\mathbf{S_2}$ with $\leq (\mathbf{1} + \varepsilon) \cdot \mathbf{k}$ mismatches

$$S_1 = T\mathbf{AAGCTTT}T,\ S_2 = C\mathbf{ACGTTTC}C,\ k = 2,\ \varepsilon = 1.5$$
$$\mathrm{LCS}_k(S_1, S_2) = 6 \Rightarrow \text{we can return } \mathbf{AGCTTT}$$

- More robust than LCS, easier to compute
- $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$ time, $\mathcal{O}(n)$ space for any $\varepsilon > 0$ **[This work]**
- Main idea: locality-sensitive hashing
- Practical: Simple system of hash functions, linear space

Reduction to the decision variant

Twenty question game with a liar



Given $\mathbf{0} \leq \mathbf{A}, \mathbf{B} \leq \mathbf{n}$. Carole must answer YES if $x \leq A$ and NO if $x > B$. To win, Paul must return some number in $[\mathbf{A}, \mathbf{B}]$.

Corollary of **[Dhagat, Gács, Winkler '92]**: For any $\mathbf{r} < \frac{1}{3}$, Paul can win by asking $\lceil \frac{8 \log \mathbf{n}}{(1-3\mathbf{r})^2} \rceil$ questions.

Decision variant

Input: integers $\mathbf{k}, \ell$, a constant $\varepsilon > \mathbf{0}$, strings $\mathbf{S_1}, \mathbf{S_2}$ of length $\mathbf{n}$

Output:

1. YES if $\ell \le \mathrm{LCS}_k$;
2. YES or NO if $\mathrm{LCS}_k < \ell \le \mathrm{LCS}_{(1+\varepsilon)k}$;
3. NO if $\mathrm{LCS}_{(1+\varepsilon)k} < \ell$.

The answer must be correct with probability at least $3/4$.

Longest Common Substring with approx. k mismatches:

- $A = \mathrm{LCS}_k$ and $B = \mathrm{LCS}_{(1+\varepsilon)k}$.
- An algorithm for the decision variant plays the role of Carole.
- With $\lceil \frac{8 \log n}{(1-3r)^2} \rceil$ questions, Paul will find $x \in [\mathrm{LCS}_k, \mathrm{LCS}_{(1+\varepsilon)k}]$ for some $1/4 < r < 1/3$.

Decision variant

Input: integers $\mathbf{k}, \ell$, a constant $\varepsilon > \mathbf{0}$, strings $\mathbf{S_1}, \mathbf{S_2}$ of length $\mathbf{n}$

Output:

1. YES if $\ell \leq \mathrm{LCS_k}$;
2. YES or NO if $\mathrm{LCS_k} < \ell \leq \mathrm{LCS_{(1+\varepsilon)k}}$;
3. NO if $\mathrm{LCS_{(1+\varepsilon)k}} < \ell$.

The answer must be correct with probability at least $3/4$.

Longest Common Substring with approx. k **mismatches:**

- $A = \mathrm{LCS_k}$ and $B = \mathrm{LCS_{(1+\varepsilon)k}}$.
- An algorithm for the decision variant plays the role of Carole.
- With $\lceil \frac{\mathbf{8} \log \mathbf{n}}{(\mathbf{1}-\mathbf{3r})^2} \rceil$ questions, Paul will find $x \in [\mathrm{LCS_k}, \mathrm{LCS_{(1+\varepsilon)k}}]$ for some $1/4 < r < 1/3$.

Locality-Sensitive Hashing

Definition: A family $\mathcal{F}$ of hash functions is called *locality-sensitive*, if for all $X, Y \in \Sigma^n$ and a hash function $h \in \mathcal{F}$ chosen u.a.r.:

- If $\text{Ham}(X, Y) \leq k$, then $h(X) = h(Y)$ with prob. $\geq p_1$;
- If $\text{Ham}(X, Y) \geq (1 + \varepsilon)k$, then $h(X) = h(Y)$ with prob. $\leq p_2$.

Main idea (simplified):
We choose a locality-sensitive hash function $h \in \mathcal{F}$ uniformly at random, and apply it to all ℓ-length substrings of S_1, S_2.

We then explore the pairs of strings that *collide*.

If there is a pair of ℓ-length substrings of X, Y with k mismatches, we will find it.

Locality-Sensitive Hashing

Definition: A family $\mathcal{F}$ of hash functions is called *locality-sensitive*, if for all $X, Y \in \Sigma^n$ and a hash function $h \in \mathcal{F}$ chosen u.a.r.:

- If $\text{Ham}(X, Y) \le k$, then $h(X) = h(Y)$ with prob. $\ge p_1$;
- If $\text{Ham}(X, Y) \ge (1 + \varepsilon)k$, then $h(X) = h(Y)$ with prob. $\le p_2$.

Main idea (simplified):
We choose a locality-sensitive hash function $h \in \mathcal{F}$ uniformly at random, and apply it to all ℓ-length substrings of S_1, S_2.

We then explore the pairs of strings that *collide*.

If there is a pair of ℓ-length substrings of X, Y with k mismatches, we will find it.

Locality-Sensitive Hashing

We construct hash functions as in **[Indyk and Motwani'98]**:

$\Pi = \{h_i, 1 \leq i \leq n : h_i(a_1 a_2 \dots a_n) = a_i\}$

$\mathcal{F} = \Pi^m$ for some parameter m

How to compute the collisions for $h \in \mathcal{F}$? We use Karp–Rabin fingerprints: $h(X) \neq h(Y) \Rightarrow \varphi(h(X)) \neq \varphi(h(Y)) \Rightarrow$ w / prob. $1 - 1/n^c$

The fingerprints can be computed in $\mathcal{O}(n \log n)$ time via FFT

Choice of parameters:

$p_1 = 1 - k/n, p_2 = 1 - (1 + \varepsilon) \cdot k/n$

$m = \log_{p_2} \lceil 1/n \rceil$

Locality-Sensitive Hashing

We construct hash functions as in **[Indyk and Motwani'98]**:

$\Pi = \{h_i, 1 \leq i \leq n : h_i(a_1 a_2 \ldots a_n) = a_i\}$

$\mathcal{F} = \Pi^m$ for some parameter m

How to compute the collisions for $h \in \mathcal{F}$? We use Karp–Rabin fingerprints: $h(X) \neq h(Y) \Rightarrow \varphi(h(X)) \neq \varphi(h(Y)) \Rightarrow$ w / prob. $1 - 1/n^c$

The fingerprints can be computed in $\mathcal{O}(n \log n)$ time via FFT

Choice of parameters:

$p_1 = 1 - k/n, p_2 = 1 - (1+\varepsilon) \cdot k/n$

$m = \log_{p_2} \lceil 1/n \rceil$

Locality-Sensitive Hashing

We construct hash functions as in **[Indyk and Motwani'98]**:

$\Pi = \{h_i, 1 \leq i \leq n : h_i(a_1 a_2 \dots a_n) = a_i\}$

$\mathcal{F} = \Pi^m$ for some parameter m

How to compute the collisions for $h \in \mathcal{F}$? We use Karp–Rabin fingerprints: $h(X) \neq h(Y) \Rightarrow \varphi(h(X)) \neq \varphi(h(Y)) \Rightarrow$ w / prob. $1 - 1/n^c$

The fingerprints can be computed in $\mathcal{O}(n \log n)$ time via FFT

Choice of parameters:

$p_1 = 1 - k/n, p_2 = 1 - (1 + \varepsilon) \cdot k/n$

$m = \log_{p_2} \lceil 1/n \rceil$

Algorithm

1: Choose a set $\mathcal{H}$ of $\Theta(n^{1/(1+\varepsilon)})$ functions from Π^m u.a.r.
2: $\mathbf{C_l^{\mathcal{H}}} :=$ set of all collisions of l-length substrings of $\mathbf{S_1}, \mathbf{S_2}$ under the hash functions in $\mathcal{H}$
3: Draw a collision $(\mathbf{X}, \mathbf{Y}) \in \mathbf{C}_\ell^{\mathcal{H}}$ uniformly at random
4: **if** $\mathrm{Ham}(\mathbf{X}, \mathbf{Y}) \leq (\mathbf{1} + \varepsilon) \cdot \mathbf{k}$ **then return** YES
5: Choose a subset $\mathbf{C'} \subseteq \mathbf{C_l^{\mathcal{H}}}$ of size $\min\{\mathbf{C}_\ell^{\mathcal{H}}, \mathbf{4nL}\}$
6: **for** $(\mathbf{X}, \mathbf{Y}) \in \mathbf{C'}$ **do**
7:     **if** $\mathrm{Ham}(\mathbf{S_1}, \mathbf{S_2}) \leq \mathbf{k}$ **then return** YES
8: **return** NO

Running time $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$:

1. Compute the hash values and C': $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$ time (FFT)
2. Pick a random collision: $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ time (reservoir sampling)
3. Test in line 5: $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time (dimension reduction)
4. Test in line 7: $\mathcal{O}(n)$ time (character-by-character)

Algorithm

```
1: Choose a set H of Θ(n^{1/(1+ε)}) functions from Π^m u.a.r.
2: C_l^H := set of all collisions of l-length substrings of S_1, S_2 under the
   hash functions in H
3: Draw a collision (X, Y) ∈ C_ℓ^H uniformly at random
4: if Ham(X, Y) ≤ (1 + ε) · k then return YES
5: Choose a subset C' ⊆ C_l^H of size min{C_ℓ^H, 4nL}
6: for (X, Y) ∈ C' do
7:     if Ham(S_1, S_2) ≤ k then return YES
8: return NO
```

Running time $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$:

1. Compute the hash values and C': $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$ time (FFT)
2. Pick a random collision: $\mathcal{O}(n^{1+1/(1+\varepsilon)})$ time (reservoir sampling)
3. Test in line 5: $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$ time (dimension reduction)
4. Test in line 7: $\mathcal{O}(n)$ time (character-by-character)

Experiments

None of the previous solutions have been implemented.

The only algorithm that seemed to be practical enough is the dynamic programming one **[Flouri et al.'15]**

We compared our algorithm with the dynamic programming one

- On random strings;
- On strings extracted from E. coli.

Lengths from 5000 to 60000, $k = 10, 25, 50$

Experiments

None of the previous solutions have been implemented.

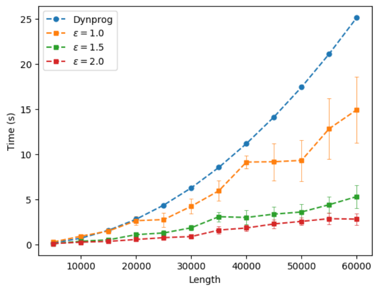
The only algorithm that seemed to be practical enough is the dynamic programming one **[Flouri et al.'15]**

We compared our algorithm with the dynamic programming one

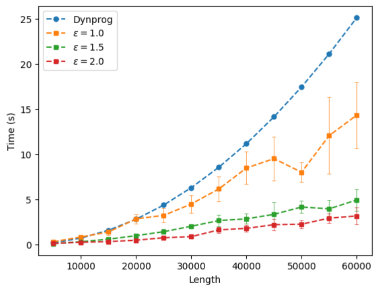
- On random strings;
- On strings extracted from E. coli.

Lengths from 5000 to 60000, $k = 10, 25, 50$

Running time



(a) Random, $k = 25$



(b) E. coli, $k = 25$

- For each length, we performed 10 independent experiments
- Big standard deviation for $\varepsilon = 1$, negligible for $\varepsilon = 1.5$ and $\varepsilon = 2.0$
- Gain up to a factor of 10 on strings of length 60000

Distortion and accuracy

We estimate distortion by computing two values:

$$r_{\min}(\varepsilon, k) = \min_{S_1,S_2}(\mathrm{LCS}_{\tilde{\mathrm{k}}}(S_1, S_2)/\mathrm{LCS}_{\mathrm{k}}(S_1, S_2))$$
$$r_{\max}(\varepsilon, k) = \max_{S_1,S_2}(\mathrm{LCS}_{\tilde{\mathrm{k}}}(S_1, S_2)/\mathrm{LCS}_{\mathrm{k}}(S_1, S_2))$$

Furthermore, we can only err by returning a string shorter than $\mathrm{LCS}_{\mathrm{k}}$.

	Random					
	$\varepsilon = 1.0$		$\varepsilon = 1.5$		$\varepsilon = 2.0$	
$k = 10$	0.92	1.50	1.00	1.53	1.13	1.87
	err = 7%		err = 0%		err = 0%	
$k = 25$	1.10	1.48	1.30	1.70	1.55	2.11
	err = 0%		err = 0%		err = 0%	

	E. coli					
	$\varepsilon = 1.0$		$\varepsilon = 1.5$		$\varepsilon = 2.0$	
$k = 10$	0.86	1.41	0.91	1.47	0.95	1.71
	err = 34%		err = 13%		err = 8%	
$k = 25$	0.94	1.45	0.96	1.75	0.98	1.96
	err = 7%		err = 5%		err = 2%	

Conclusion

- Longest common substring with k mismatches cannot be solved in subquadratic time unless SETH is false
- New approximation algorithm solves the problem in $\mathcal{O}(\mathbf{n}^{1+1/(1+\varepsilon)} \log^3 \mathbf{n})$ time and $\mathcal{O}(\mathbf{n})$ space
- Simple and practical — faster than the dynamic programming solution for $\varepsilon > 1$
- Small distortion compared to LCS_k (even though no theoretical guarantee)
- Good accuracy

Thank you!