Randomized and Parameterized Algorithms for the Closest String Problem

Zhi-Zhong Chen

Tokyo Denki University

Joint with

B. Ma : University of Waterloo

L. Wang : City University of Hong Kong

The Closest String Problem -- Definition

Given: Several strings and an integer d .

For example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$d = 3$

Output: A string t of length L such that the Hamming distance between t and each input string is at most d.

For example: t = GAGCGGCGTAG

The Closest String Problem -- Definition

Given: Several strings and an integer d .

For example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$d = 3$

Output: A string t of length L such that the Hamming distance between t and each input string is at most d.

For example: t = GAGCGGCGTAG

The Closest String Problem -- Applications

- Motif detection
- Finding unbiased consensus of a protein family
- Universal PCR primer design
- Genetic drug target identification
- Genetic probe design

…

The Closest String Problem – Previous works

Many papers have been published in journals or conference proceedings on bioinformatics, computational biology, and computer science.

J. ACM
SIAM Journal on Computing
STOC
FOCS
…

- **Exact algorithms (i.e., some of them are fixed-parameter algorithms)** } **Our focus here!**
- **Approximation algorithms** → **Objective: Minimize *d* (called the radius).**

The Closest String Problem -- previous FP algorithms

- Take d as the parameter and strive to design algorithms that run in time exponential in d only (i.e., in time polynomial in L and the total number of input strings).

- Gramm, et al. 2003 (Algorithmica): $O(nL + nd(d+1)^d)$ time.
  - n : the number of input strings.

- Boucher and Brown, 2009 (BIC):

$$O\left(nL + nd\left(\sigma - \frac{\sigma}{L}\right)^L\right) \text{ expected time.}$$

  - σ : the alphabet size.

Gramm, et al.'s Algorithm: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GACCGGGGTTG ← **The initial candidate solution.**

1 : GGGCGGCGTCA

This t is not a solution yet.

Among the positions where the two strings disagree, arbitrarily select $d + 1$ positions

Gramm, et al.'s Algorithm: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GACCGGGGTTG ← **The initial candidate solution.**

1 : GGGCGGCGTCA

This t is not a solution yet.

Among the positions where the two strings disagree, arbitrarily select $d + 1$ positions (say, the leftmost $d + 1$ positions). At least one of the selected positions must be modified to agree with the other string but we don't know which one. So, we try the $d + 1$ choices in turn.

The Closest String Problem -- previous FP algorithms

- **Boucher and Brown, 2009 (BIC):**

$$O\left(nL + nd\left(\sigma - \frac{\sigma}{L}\right)^{L}\right)$$ **expected time.**

- **σ : the alphabet size.**

Note: **This is not an FPT (fixed-parameter tractable) algorithm because its running time is exponential in L (rather than in d).**

Idea behind the algorithm:

Start with a randomly chosen candidate solution and then repeat (randomly) modifying it until a solution is obtained.

Similar to

- **Papadimitriou's randomized algorithm for 2-SAT, and**
- **Schoning's randomized algorithm for k-SAT.**

Boucher & Brown's Algorithm: example

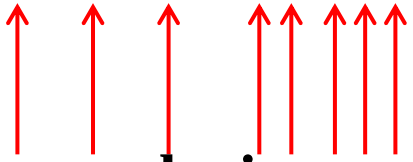
Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

t = AGTCCGTAAGC ← **A randomly chosen candidate solution.**

1 : GGGCGGCGTCA



This t is not a solution yet, because some input strings have distance larger than d to t.

Among the input strings with distance $>d$ to t, select one uniformly at random. Among the positions where t and the selected string disagree, select one uniformly at random and modify the letter of t at the position to agree with the selected string.

Boucher & Brown's Algorithm: example

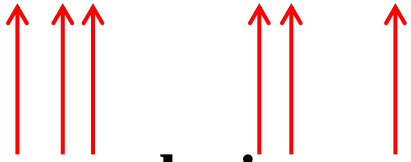
Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

t = AGTCCGTATGC ← **The candidate solution after 1 modification.**

8 : GAGCCGAGTGG



This t is not a solution yet, because some input strings have distance larger than d to t.

Among the input strings with distance $>d$ to t, select one uniformly at random. Among the positions where t and the selected string disagree, select one uniformly at random and modify the letter of t at the position to agree with the selected string.

Boucher & Brown's Algorithm: example

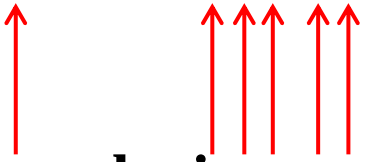
Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

t = AGGCCGTATGC ← **The candidate solution after 2 modifications.**

5 : AAGCCGCGAAG



This t is not a solution yet, because some input strings have distance larger than d to t.

Among the input strings with distance $>d$ to t, select one uniformly at random. Among the positions where t and the selected string disagree, select one uniformly at random and modify the letter of t at the position to agree with the selected string.

Boucher & Brown's Algorithm: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

t = AGGCCGTAAGC ← **The candidate solution after 3 modifications.**

1 : GGGCGGCGTCA

This t is not a solution yet, because some input strings have distance larger than d to t.

Among the input strings with distance $>d$ to t, select one uniformly at random. Among the positions where t and the selected string disagree, select one uniformly at random and modify the letter of t at the position to agree with the selected string.

Boucher & Brown's Algorithm: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$

t = AGGCCGTATGC ← **The candidate solution after 4 modifications.**

This t is not a solution yet, because some input strings have distance larger than d to t.

Repeat the process enough times (namely, $\Omega(L)$).
If no solution is found after the repetitions, then give up the current t and restart the whole process from scratch.

The Closest String Problem -- previous FP algorithms

- **Ma & Sun, 2008 (RECOMB, SIAM J. on Comput.):**

  $O(nL + nd \cdot (16(\sigma - 1))^d)$ **time.**

  - σ **: the alphabet size.**

- $\sigma = 2$ **(binary):** $O(nL + nd \cdot 16^d)$ **time.**
- $\sigma = 4$ **(DNA):** $O(nL + nd \cdot 48^d)$ **time.**
- $\sigma = 20$ **(protein):** $O(nL + nd \cdot 304^d)$ **time.**

This is the first algorithm which runs in polynomial time for constant-sized alphabets and logarithmic-valued *d*.

The idea is to try and modify multiple positions of *t* in each repetition.

Nontrivial algorithm and analysis!

The Closest String Problem -- previous FP algorithms

- **Wang & Zhu, 2009 (FAW):**

$$O\left(nL + nd \cdot \left(2^{3.25}(\sigma - 1)\right)^d\right) \text{ time.}$$

  - $\sigma = 2$ (binary): $O\left(nL + nd \cdot 9.5^d\right)$ time.
  - $\sigma = 4$ (DNA): $O\left(nL + nd \cdot 29^d\right)$ time.
  - $\sigma = 20$ (protein): $O\left(nL + nd \cdot 181^d\right)$ time.

Refining Ma & Sun's algorithm

- **Chen & Wang, 2011 (TCBB):**

$$O\left(nL + nd \cdot \left(\sqrt{2}\,\sigma - \sqrt[4]{8}\,(\sqrt{2}+1)(1+\sqrt{\sigma - 1}) - 2\sqrt{2}\right)^d\right) \text{ time.}$$

  - $\sigma = 2$ (binary): $O\left(nL + nd \cdot 8^d\right)$ time (by different analysis).
  - $\sigma = 4$ (DNA): $O\left(nL + nd \cdot 13.92^d\right)$ time.
  - $\sigma = 20$ (protein): $O\left(nL + nd \cdot 47.21^d\right)$ time.

The Closest String Problem -- previous FP algorithms

- **Chen, Ma, & Wang, 2012 (JCSS):**

  $O\left(nL + nd \cdot \left(1.612(\sigma + \beta^2 + \beta - 2)\right)^d\right)$ time, where

  $\beta = \alpha^2 + 1 - 2\alpha^{-1} + \alpha^{-2}$ and $\alpha = \sqrt[3]{\sqrt{\sigma - 1} + 1}$

  - $\sigma = 2$ (binary): $O\left(nL + nd^3 \cdot 6.74^d\right)$ time (by different analysis)
  - $\sigma = 4$ (DNA): $O\left(nL + nd \cdot 13.19^d\right)$ time.
  - $\sigma = 20$ (protein): not better than the previous best.

Based on a novel 3-string approach: Instead of using a single input string to guide the selection of positions of *t* to modify, use two input strings to guide the selection.

Very nontrivial algorithm and complicated analysis!

The Simplest New Algorithm (binary case): example

Example:

```
1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101
```

$L = 11$
$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

```
t = 000100000101
1 : 001001000100
```

The initial candidate solution.

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

Among the positions where the two strings disagree, select one uniformly at random and flip the bit of t at the selected position.

The Simplest New Algorithm (binary case): example

Example:

```
1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101
```

$L = 11$

$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

```
t = 00000000101
3 : 11000000010
```

The candidate solution after 1 modification.

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

Among the positions where the two strings disagree, select one uniformly at random and flip the bit of t at the selected position.

The Simplest New Algorithm (binary case): example

Example:

```
1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101
```

$L = 11$

$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

```
t = 00000000011
3 : 11000000010
```

The candidate solution after 2 modifications.

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

Among the positions where the two strings disagree, select one uniformly at random and flip the bit of t at the selected position.

The Simplest New Algorithm (binary case): example

Example:

1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101

$L = 11$
$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = 010000000111 ← **The candidate solution after 3 modifications.**

1 : 001001000100

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

However, we already modified d positions of t.
So, we give up this t and restart the whole process from scratch.

The Simplest New Algorithm (binary case): example

Example:

```
1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101
```

$L = 11$

$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

```
t = 000100000101
1 : 001001000100
```

The initial candidate solution.

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

Among the positions where the two strings disagree, select one uniformly at random and flip the bit of t at the selected position.

The Simplest New Algorithm (binary case): example

Example:

```
1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101
```

$L = 11$

$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

```
t = 000000000101
3 : 110000000010
```

The candidate solution after 1 modification.

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

Among the positions where the two strings disagree, select one uniformly at random and flip the bit of t at the selected position.

The Simplest New Algorithm (binary case): example

Example:

```
1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101
```

$L = 11$

$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

```
t = 000000000100
5 : 101000100000
```

The candidate solution after 2 modifications.

This t is not a solution yet, because there is an input string whose Hamming distance to t is larger than d.

Among the positions where the two strings disagree, select one uniformly at random and flip the bit of t at the selected position.

The Simplest New Algorithm (binary case): example

Example:

1 : 001001000100
2 : 011000000001
3 : 110000000010
4 : 001000000101
5 : 101000100000
6 : 001100000010
7 : 001010000100
8 : 001000110000
9 : 000010010100
10: 000100000101

$L = 11$
$d = 3$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

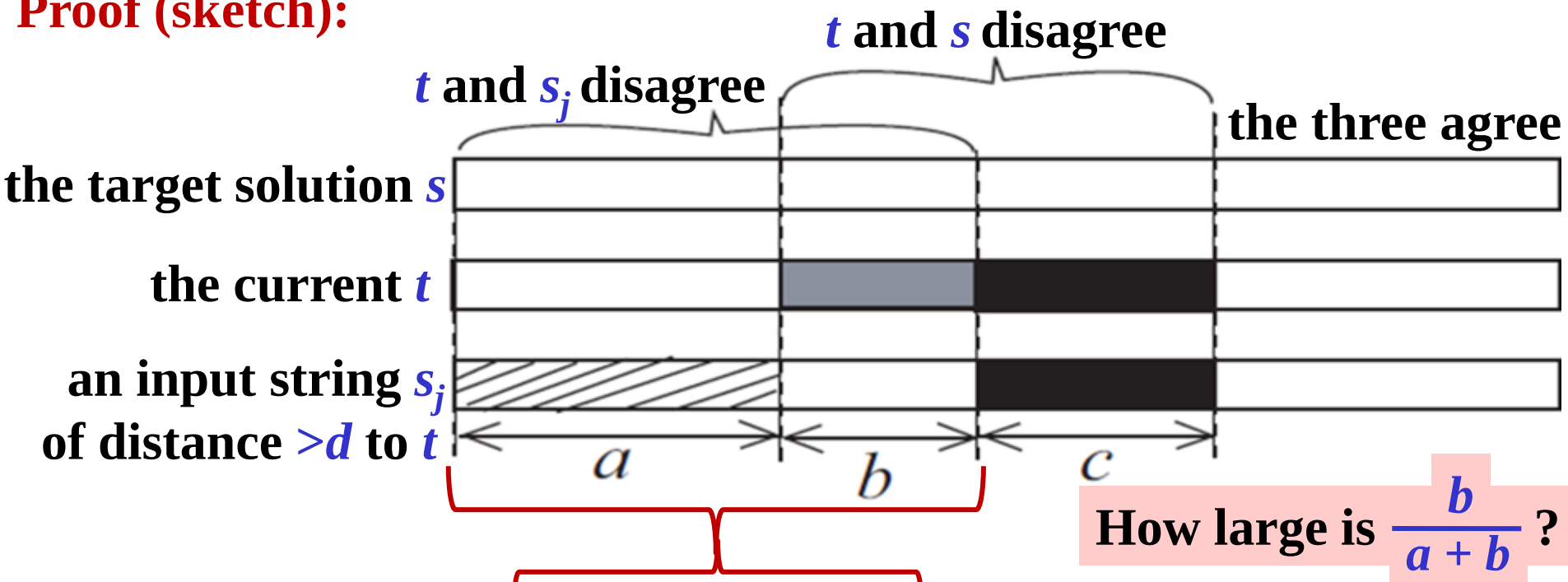
t = 000000000000 ← **The candidate solution after 3 modifications.**

This t is now a solution,
because the Hamming distance between t and every input string is at most d.

The Simplest New Algorithm (binary case): analysis

Lemma 1: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(2d)^d}$.

Proof (sketch):



How large is $\frac{b}{a+b}$?

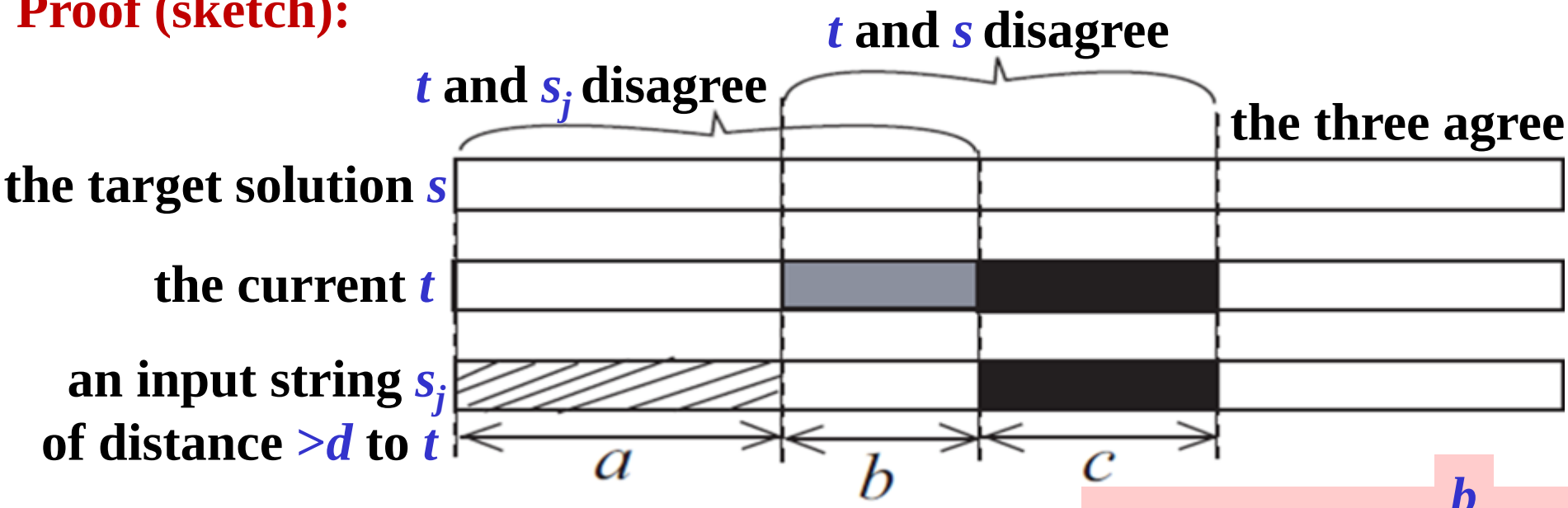
We flip one of the $a + b$ positions uniformly at random if t is still not a solution.

We are lucky if the selected position is among the b positions.

The Simplest New Algorithm (binary case): analysis

Lemma 1: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(2d)^d}$.

Proof (sketch):



How large is $\frac{b}{a+b}$?

$d \geqq a + c$ ($\because s$ is a solution)

$a + b = d + k$ ($\because t$ is still not a solution $\therefore k > 0$)

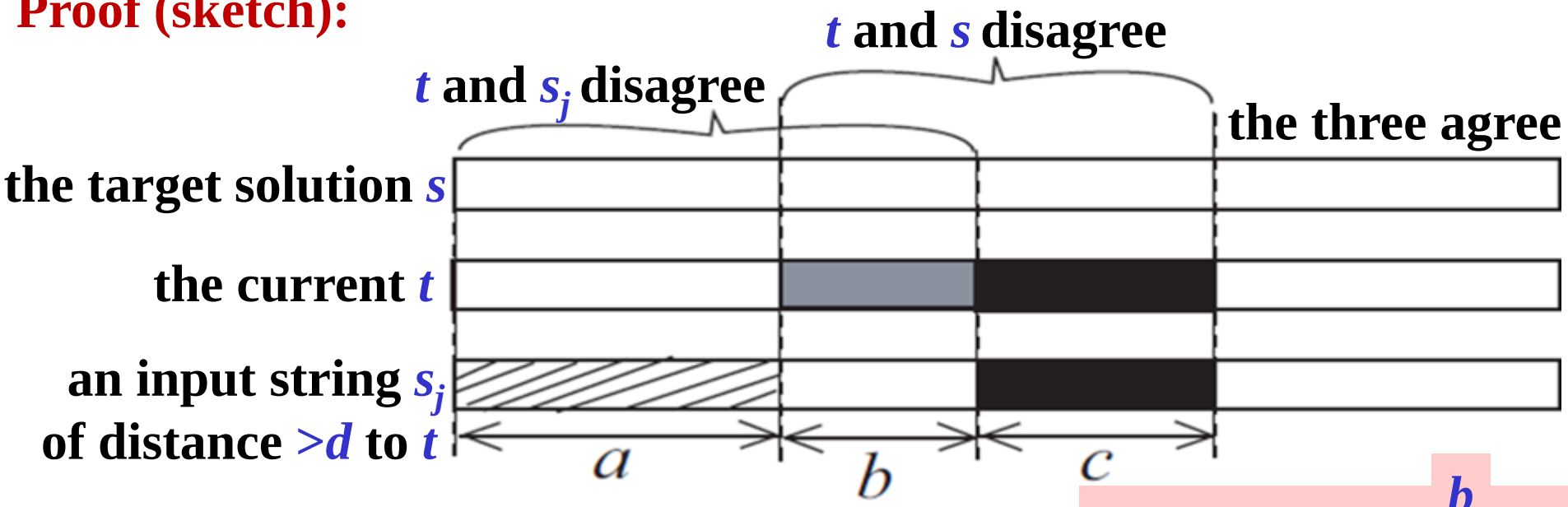
$b + c = d_1 - (j - 1)$ (if we have made $j - 1$ modifications to t and we have been lucky in the $j - 1$ modifications)

d_1 is the Hamming distance between s and the initial t.

The Simplest New Algorithm (binary case): analysis

Lemma 1: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(2d)^d}$.

Proof (sketch):



$d \geqq a + c$

$a + b = d + k$

$b + c = d_1 - (j - 1)$

$\Rightarrow 2b \geqq d_1 - (j - 1) + k$

$$\frac{b}{a+b} \geqq \frac{d_1 - (j-1) + k}{2(d+k)} \geqq \frac{d_1 - (j-1)}{2d}$$

How large is $\frac{b}{a+b}$?

The Simplest New Algorithm (binary case): analysis

Lemma 1: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(2d)^d}$.

Proof (sketch):

The probability that we are lucky all the way is at least

$$\frac{d_1}{2d} \cdot \frac{d_1 - 1}{2d} \cdot \frac{d_1 - 2}{2d} \cdot \cdots \cdot \frac{1}{2d} = \frac{d_1!}{(2d)^{d_1}} \geqq \frac{d!}{(2d)^d}$$

$(\because d_1 \leqq d)$

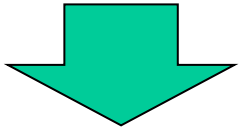
Q.E.D.

If we have made $j - 1$ modifications to t and we have been lucky in the $j - 1$ modifications, then with probability at least $\frac{d_1 - (j-1)}{2d}$ we are lucky in the j-th modification as well.

$$\frac{b}{a+b} \geqq \frac{d_1 - (j-1) + k}{2(d+k)} \geqq \frac{d_1 - (j-1)}{2d}$$

The Simplest New Algorithm (binary case): analysis

Unfortunately, if there is no solution, then the repetition never ends.



We want to gamble on "no solution" after repeating enough times.

In this way, we only make one-sided error:

- **In case there is no solution, we never report a solution and hence never make an error.**
- **In case there is a solution, we may report "no solution" and hence make an error.**

Lemma 1 ensures that we fail to find a solution in 1 repetition,

with probability $\leqq 1 - \frac{d!}{(2d)^d} \fallingdotseq 1 - \frac{\sqrt{2\pi d}}{(2e)^d}$

$\therefore$ **We fail to find a solution in r repetitions,**

with probability $\leqq \left(1 - \frac{\sqrt{2\pi d}}{(2e)^d}\right)^r \leqq e^{-\frac{\sqrt{2\pi d}}{(2e)^d} \cdot r}$

The Simplest New Algorithm (binary case): analysis

$\therefore$ We can stop after $O\left(\frac{(2e)^d}{\sqrt{2\pi d}}\right)$ repetitions.

$\therefore$ We fail to find a solution in $10 \cdot \frac{(2e)^d}{\sqrt{2\pi d}}$ repetitions,

with probability $\leqq e^{-10} \leqq 0.005\%$

$\therefore$ We fail to find a solution in r repetitions,

with probability $\leqq \left(1 - \frac{\sqrt{2\pi d}}{(2e)^d}\right)^r \leqq e^{-\frac{\sqrt{2\pi d}}{(2e)^d} \cdot r}$

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GACCGGGGTTG ← **The initial candidate solution.**

1 : GGGCGGCGTCA

This t is not a solution yet.

Among the positions where the two strings disagree, select one uniformly at random, change the letter of t at the selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GAGCGGGGTTG ← **The candidate solution after 1 modification.**

1 : GATCGGCTGAG

This t is not a solution yet.

Among the positions where the two strings disagree, select one uniformly at random, change the letter of t at the selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GAGCGGAGTTG ← **The candidate solution after 2 modifications.**

2 : GATCGGCTGAG

This t is not a solution yet.

Among the positions where the two strings disagree, select one uniformly at random, change the letter of t at the selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GAGCGGAGGTG ← **The candidate solution after 3 modifications.**

2 : GATCGGCTGAG

This t is not a solution yet.

However, we already modified d positions of t.
So, we give up this t and restart the whole process from scratch.

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GACCGGGGTTG ← **The initial candidate solution.**

1 : GGGCGGCGTCA

This t is not a solution yet.

Among the positions where the two strings disagree, select one uniformly at random, change the letter of t at the selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GACCGGGGTAG ← **The candidate solution after 1 modification.**

1 : GGGCGGCGTCA

This t is not a solution yet.

Among the positions where the two strings disagree, select one uniformly at random, change the letter of t at the selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

Extend the Simplest New Algorithm to general alphabets

Example:

```
1:  GGGCGGCGTCA
2:  GATCGGCTGAG
3:  CACCGGCGTAG
4:  GAGCGGCGGGC
5:  AAGCCGCGAAG
6:  ACTCGGCGTAG
7:  GAAAGGCCTAG
8:  GAGCCGAGTGG
9:  GAGATGCGTAC
10: GACCGGGGTTG
```

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

t = GAGCGGGGTAG ← **The candidate solution after 2 modifications.**

1 : GAGATGCGTAC

This t is not a solution yet.

Among the positions where the two strings disagree, select one uniformly at random, change the letter of t at the selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

Extend the Simplest New Algorithm to general alphabets

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

$L = 11$
$d = 3$
$\sigma = 4$

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

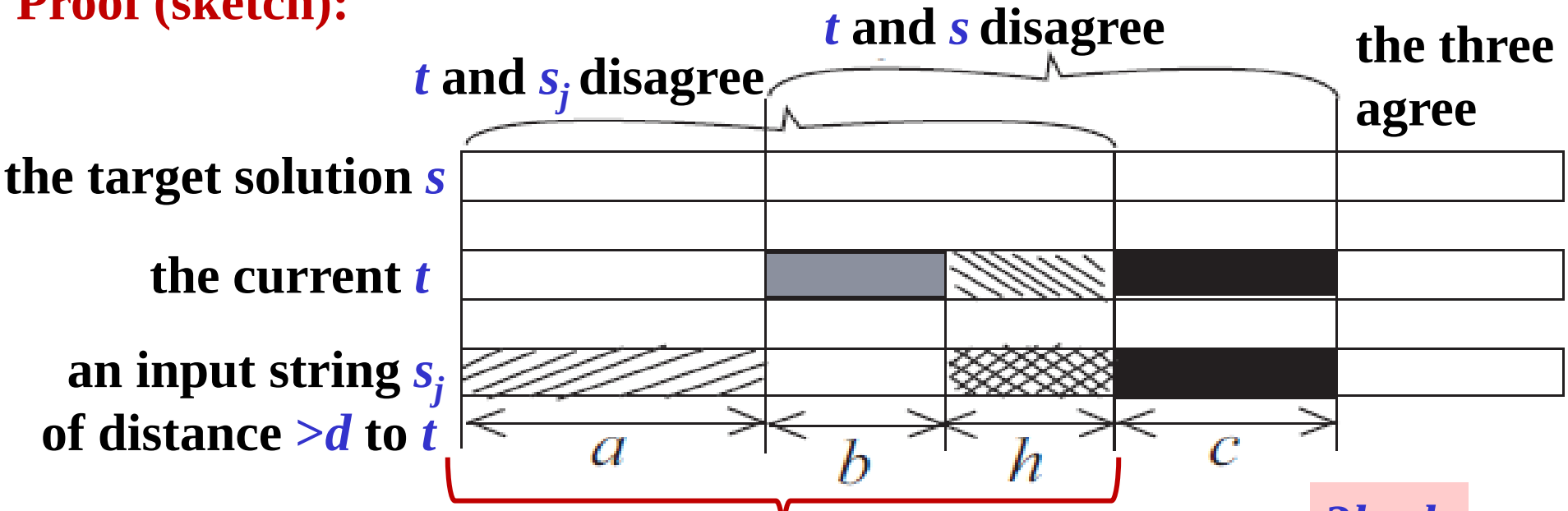
t = GAGCGGCGTAG ← **The candidate solution after 3 modifications.**

This t is now a solution,
because the Hamming distance between t and every input string is at most d.

The Simplest New Algorithm (general case): analysis

Lemma 2: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(\sigma d)^d}$.

Proof (sketch):



How large is $\frac{2b+h}{\sigma(a+b+h)}$?

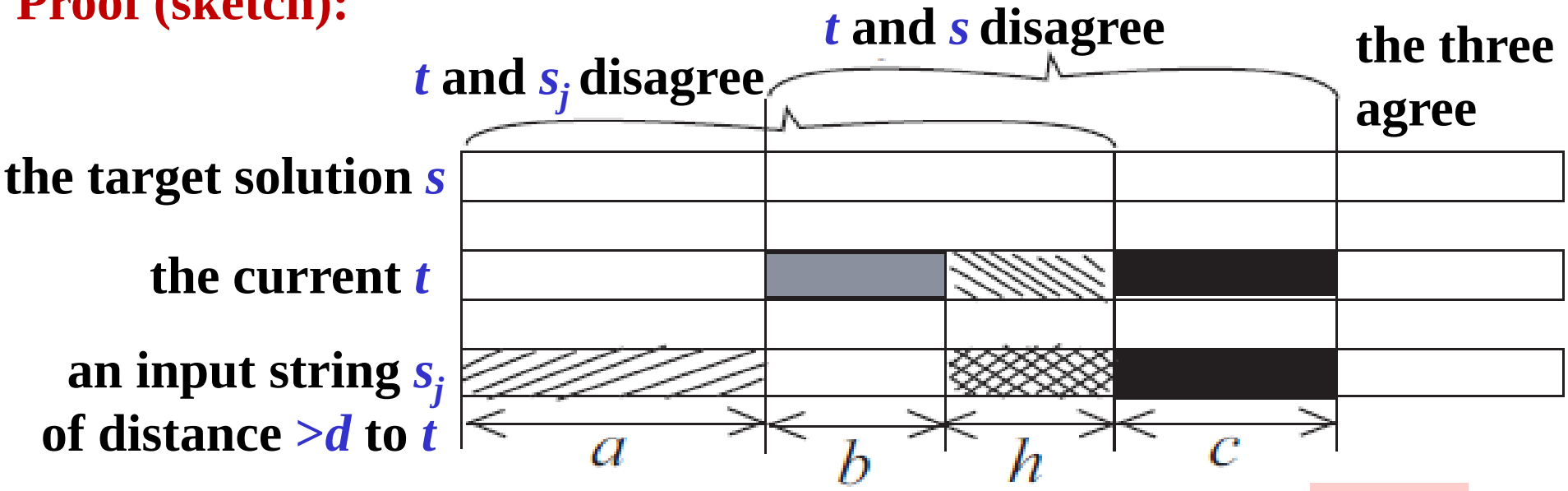
We select one of the $a + b + h$ positions uniformly at random and modify the position non-uniformly, if t is still not a solution.

We are lucky if the selected position is among the $b+h$ positions and is modified to the letter of t.

The Simplest New Algorithm (general case): analysis

Lemma 2: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(\sigma d)^d}$.

Proof (sketch):



$d \geqq a + c + h$ ($\because s$ is a solution)

$a + b + h = d + k$ ($\because t$ is still not a solution $\therefore k > 0$)

$b + c + h = d_1 - (j - 1)$ (if we have made $j - 1$ modifications to t and we have been lucky in the $j - 1$ modifications)

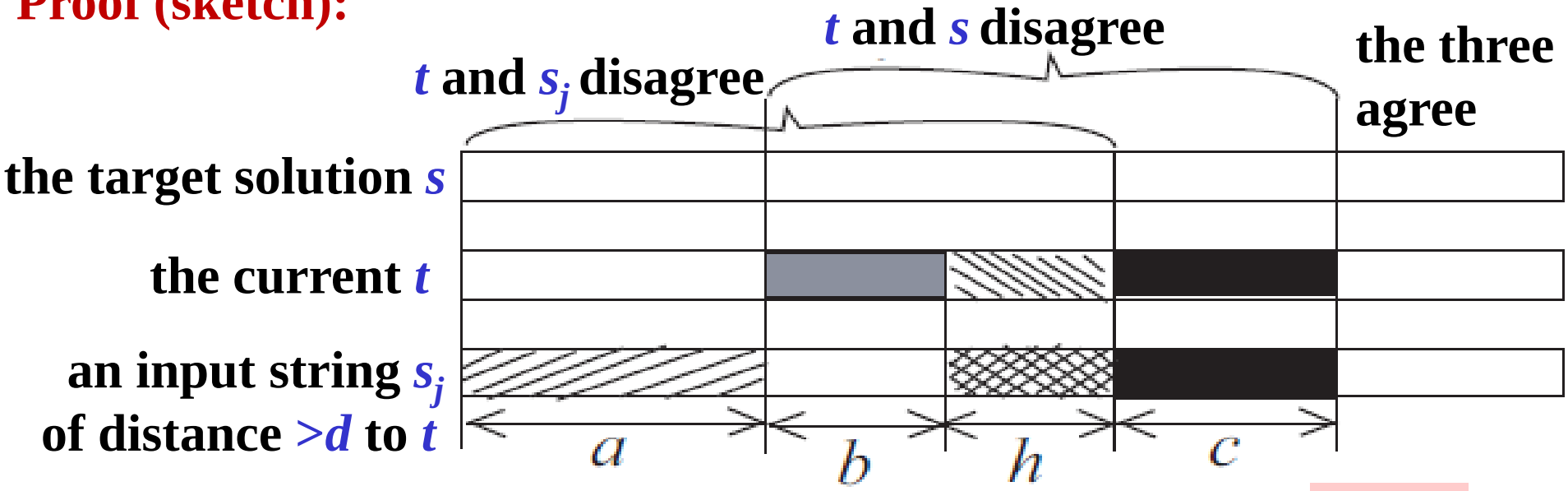
d_1 is the Hamming distance between s and the initial t.

How large is $\frac{2b + h}{\sigma(a + b + h)}$?

The Simplest New Algorithm (general case): analysis

Lemma 2: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(\sigma d)^d}$.

Proof (sketch):



$d \geqq a + c + h$

$a + b + h = d + k$

$b + c + h = d_1 - (j - 1)$

$2b + h \geqq d_1 - (j - 1) + k$

How large is $\frac{2b + h}{\sigma(a + b + h)}$?

$$\frac{2b + h}{\sigma(a + b + h)} \geqq \frac{d_1 - (j - 1) + k}{\sigma(d + k)} \geqq \frac{d_1 - (j - 1)}{\sigma d}$$

The Simplest New Algorithm (general case): analysis

Lemma 2: If there is a solution s, then one repetition finds a solution with probability at least $\frac{d!}{(\sigma d)^d}$.

Proof (sketch):

The probability that we are lucky all the way is at least

$$\frac{d_1}{\sigma d} \cdot \frac{d_1 - 1}{\sigma d} \cdot \frac{d_1 - 2}{\sigma d} \cdot \cdots \cdot \frac{1}{\sigma d} = \frac{d_1!}{(\sigma d)^{d_1}} \geqq \frac{d!}{(\sigma d)^d}$$

$(\because d_1 \leqq d)$

Q.E.D.

If we have made $j - 1$ modifications to t and we have been lucky in the $j - 1$ modifications, then with probability at least $\frac{d_1 - (j-1)}{\sigma d}$ we are lucky in the j-th modification as well.

$$\frac{2b + h}{\sigma(a + b + h)} \geqq \frac{d_1 - (j-1) + k}{\sigma(d + k)} \geqq \frac{d_1 - (j-1)}{\sigma d}$$

Ideas for Improving the algorithm

Idea 1: Modify each position of t at most once.

Idea 2: When starting to modify the initial t, find an input string s_i whose Hamming distance to t is maximized.

Idea 3: Among the positions where t and s_i disagree, select $d_i - d$ positions uniformly at random, and change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$, where d_i is the Hamming distance between t and s_i .

The Improved New Algorithm (general case): example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

$t =$ GACCGGGGTTG ← **The initial candidate solution.**

This t is not a solution yet.

5 : GGGCCGCGAAG ← **The input string farthest from t.** $d_i = 5$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

The Improved New Algorithm (general case): example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGGGTGG ← **The candidate solution after 2 modifications.**

1 : GGGCGGCGTCA

This t is not a solution yet. $d_i = 4$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

The Improved New Algorithm (general case): example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGGGTGA ← **The candidate solution after 3 modifications.**

9: GAGATGCGTAC

This t is not a solution yet. $d_i = 4$

However, we already modified d positions of t.
So, we give up this t and repeat the whole process from scratch.

The Improved New Algorithm (general case): example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GACCGGGGTTG ← **The initial candidate solution.**

This t is not a solution yet.

5 : GAGCCGCGAAG ← **The input string farthest from t.** $d_i = 5$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

The Improved New Algorithm (general case): example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGCGTTG ← **The candidate solution after 2 modifications.**

2: GATCGGCTGAG

This t is not a solution yet. $d_i = 4$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

The Improved New Algorithm (general case): example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: AAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTAC
10: GACCGGGGTTG

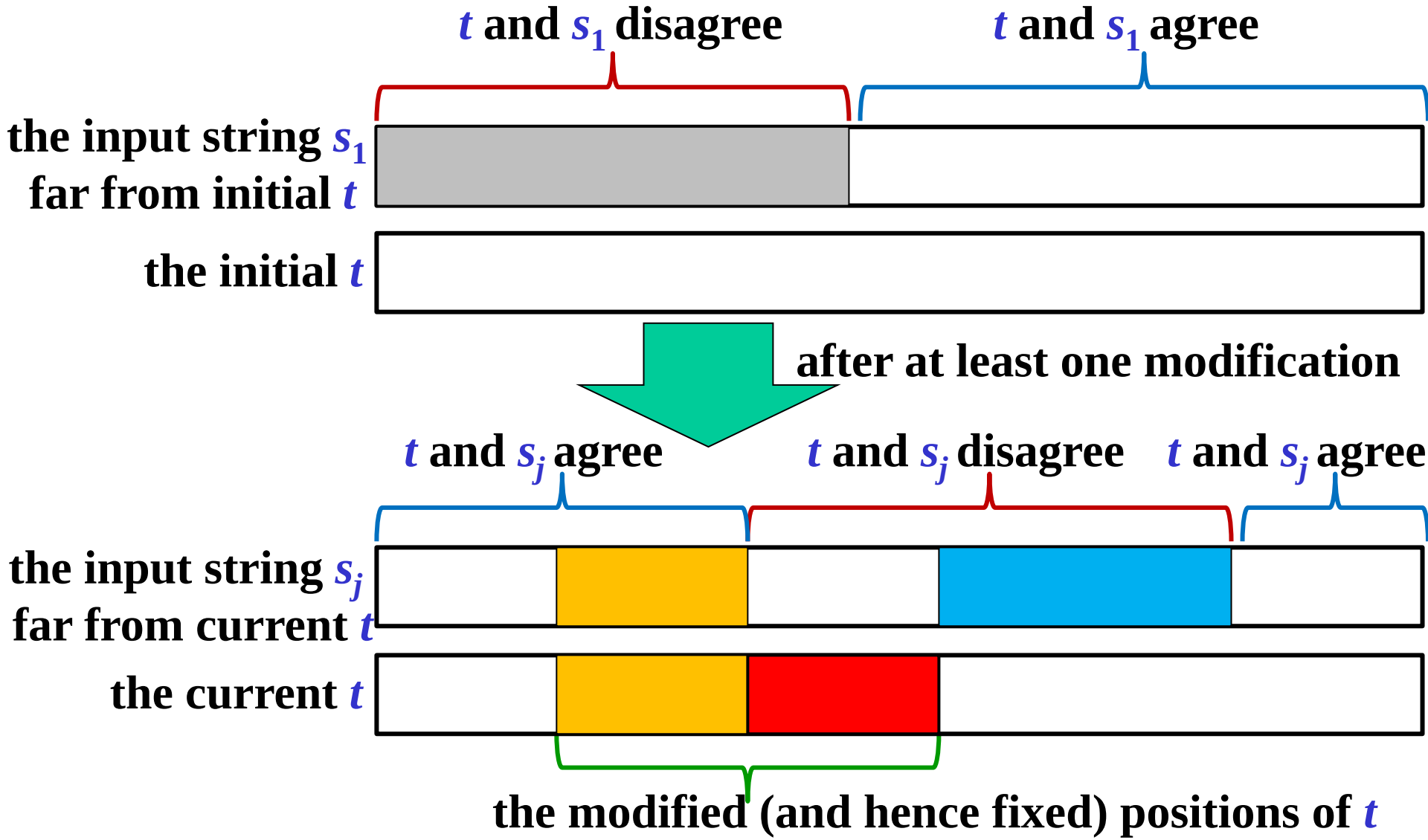
t = GAGCGGCGTAG ← **The candidate solution after 3 modifications.**

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

This t is now a solution,
because the Hamming distance between t and every input string is at most d.

The Improved New Algorithm (general case): analysis



★**A significant portion** of the positions where t and s_j disagree have been modified and hence fixed. So, we are more likely to be lucky.

The Improved New Algorithm (general case): analysis

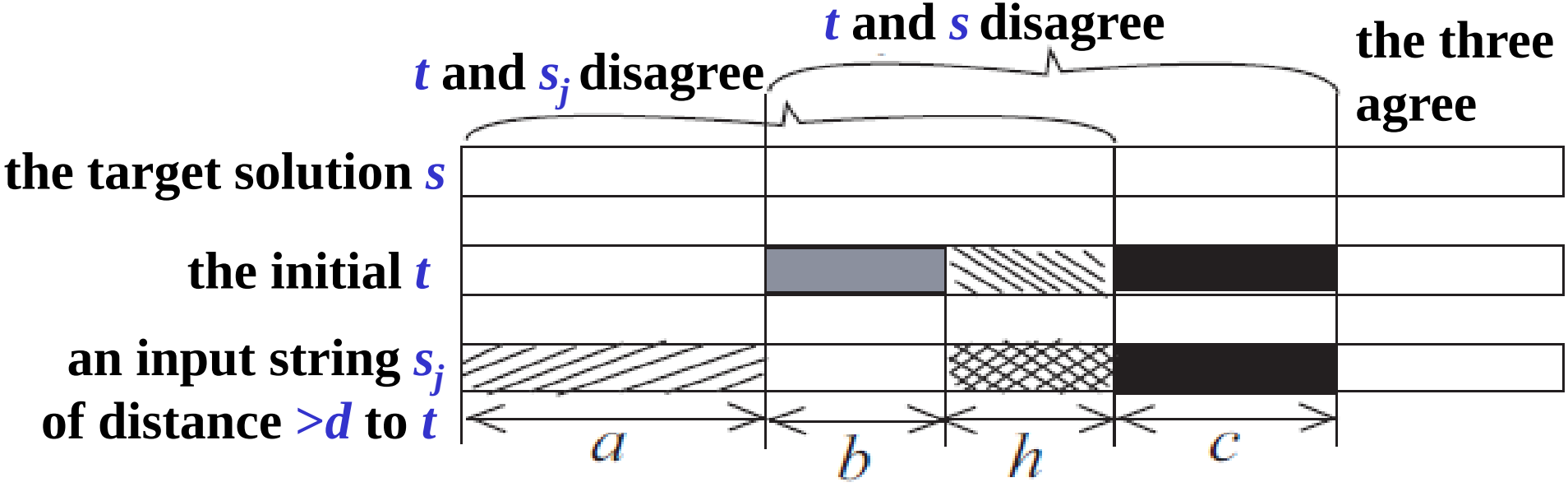
Lemma 3: If there is a solution s, then one repetition finds a solution with probability $= \Omega\left(\sqrt{d}\left(\frac{(1+\varepsilon)^{\frac{1+\varepsilon}{2}}(1-\varepsilon)^{\frac{1-\varepsilon}{2}}}{2^{1+\varepsilon}\sigma}\right)^d\right)$, where $\varepsilon = \frac{d_1 - d}{d}$ and d_1 is the largest Hamming distance between the initial t and an input string.

The probability reaches its minimum value $\Omega\left(\sqrt{d}\left(\frac{2}{5\sigma}\right)^d\right)$ at $\varepsilon = 0.6$.

$O\left(nL + n\sqrt{d} \cdot (2.5\sigma)^d\right)$ expected time.

The algorithm is fast for small σ.

A Better Algorithm for Large Alphabets: idea



Lemma 3: If there is a solution s, then $b \geqq d_i - d$, where d_i is the Hamming distance between t and s_j.

Proof (sketch): $d \geqq a + c + h$ ($\because s$ is a solution)

$a + b + h = d_i$ (by the definition of d_i)

$\Rightarrow b \geqq d_i - d$

Q.E.D.

$\therefore$ Among the d_i positions where the two strings disagree, at least $d_i - d$ positions of t must be modified to agree with s_j.

If σ is large, we are luckier by randomly guessing the $d_i - d$ positions.

A Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GACCGGGGTTG ← **The initial candidate solution.**

This t is not a solution yet.

9: GAGATGCGTTC ← **The input string farthest from t.** $d_i = 5$

Among the d_i positions where the two strings disagree, at least $d_i - d$ positions of t must be modified to agree with the other string. So, we select $d_i - d$ positions uniformly at random among the d_i positions, modify t at the selected positions to agree with the other string.

A Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGCGTTG ← **The candidate solution after 2 modifications.**

2: GATCGGCTGAG

This t is not a solution yet. $d_i = 4$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

A Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGCGTCG ← **The candidate solution after 3 modifications.**

9: GAGATGCGTTC

This t is not a solution yet. $d_i = 4$

However, we already modified d positions of t.
So, we give up this t and repeat the whole process from scratch.

A Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GACCGGGGTTG ← **The initial candidate solution.**

This t is not a solution yet.

9: GAGATGCGTTC ← **The input string farthest from t.**

$d_i = 5$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, modify t at the selected positions to agree with the other string.

A Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGCGTTG ← **The candidate solution after 2 modifications.**

2: GATCGGCTGAG

This t is not a solution yet. $d_i = 4$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

A Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

t = GAGCGGCGTAG ← **The candidate solution after 3 modifications.**

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

This t is now a solution, because the Hamming distance between t and every input string is at most d.

A Better Algorithm for Large Alphabets: analysis

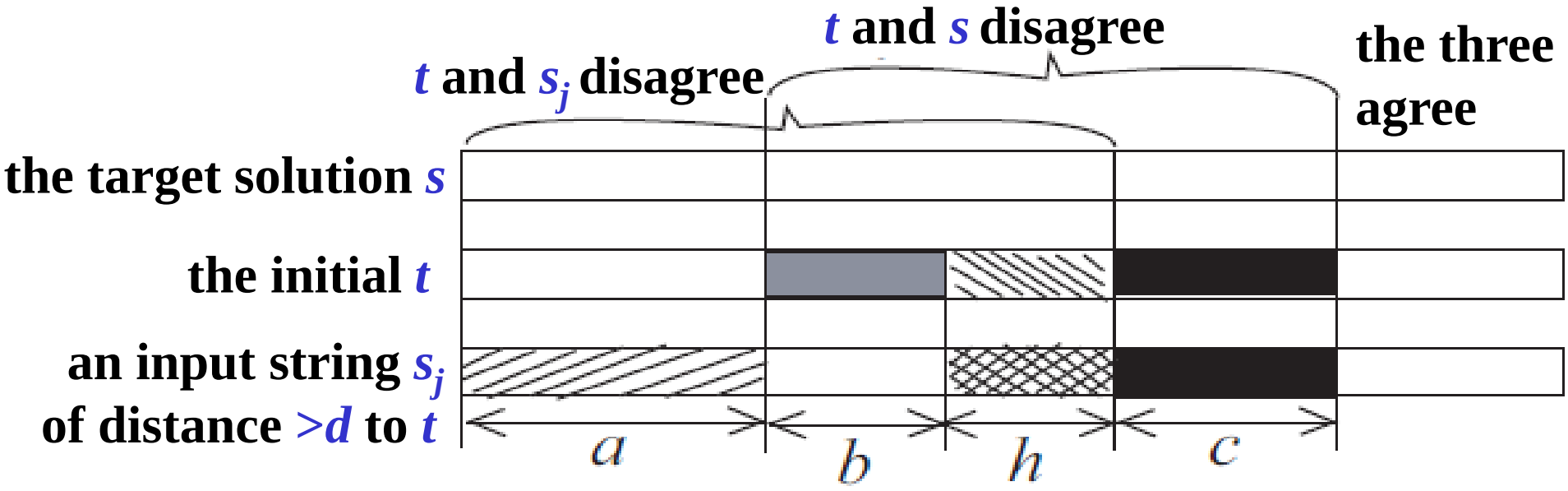
Lemma 4: If there is a solution s, then one repetition finds a solution with probability $= \Omega\left(d\sqrt{\varepsilon(1-\varepsilon)}\left(\frac{\varepsilon^{\varepsilon}(1-\varepsilon)^{1-\varepsilon}}{2^{1+\varepsilon}\sigma^{1-\varepsilon}}\right)^{d}\right)$, where $\varepsilon = \frac{d_1 - d}{d}$ and d_1 is the largest Hamming distance between the initial t and an input string.

The probability reaches its minimum value $\Omega\left(\sqrt{d}\left(\frac{1}{2\sigma+4}\right)^{d}\right)$ at $\varepsilon = \frac{2}{\sigma+2}$

$O\left(nL + n\sqrt{d}\cdot(2\sigma+4)^{d}\right)$ expected time.

The algorithm is fast for relatively large σ.

An Even Better Algorithm for Large Alphabets: idea



Lemma 3: If there is a solution s, then $b \geqq d_i - d$, where d_i is the Hamming distance between t and s_j.

Proof (sketch): $d \geqq a + c + h$ ($\because s$ is a solution)

$a + b + h = d_i$ (by the definition of d_i)

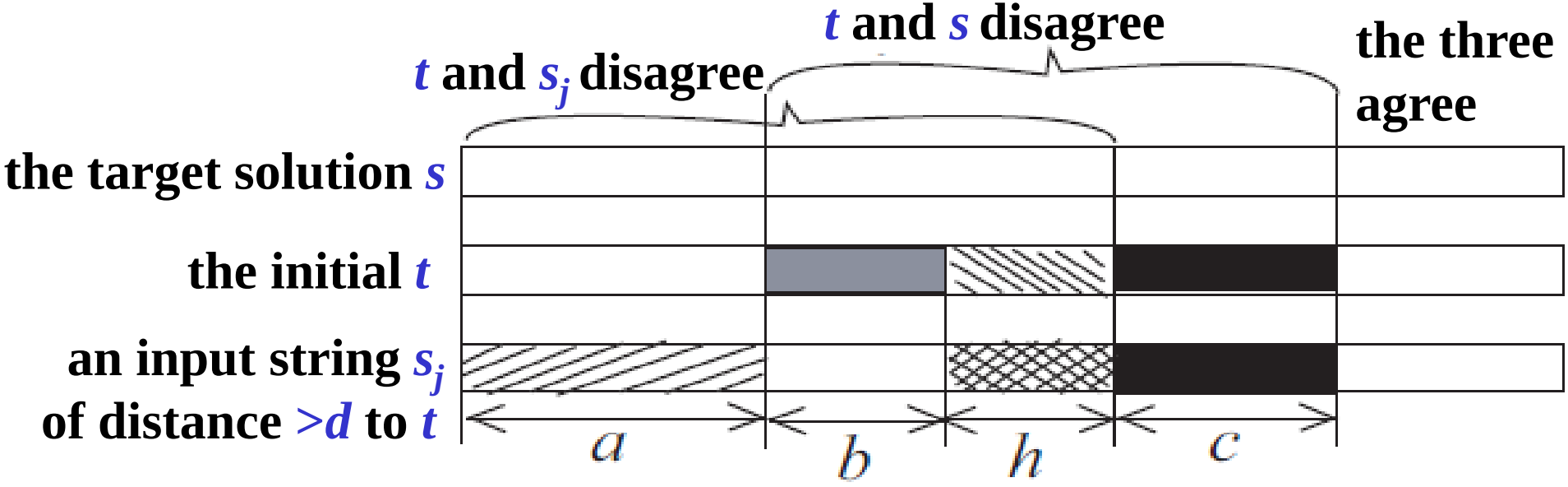
$\Rightarrow b \geqq d_i - d$

Q.E.D.

$c \leqq b - (d_i - d)$

$\therefore c \leqq \min\{d - b - h, b - (d_i - d)\}$

An Even Better Algorithm for Large Alphabets: idea

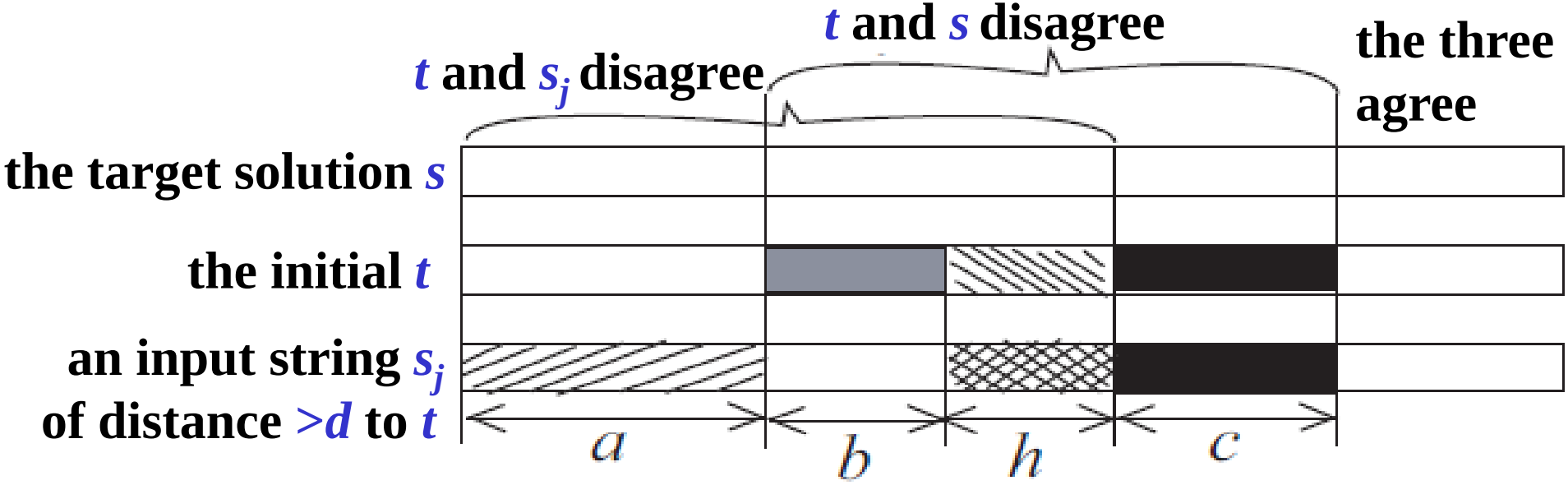


We find b and the b positions, and
modify t at the b positions to agree with s_j;
further find h, the h positions, and the letters of t at the h positions,
and modify t at the h positions accordingly.

Then, it remains to modify at most $\min\{d - b - h, b - (d_i - d)\}$ positions of t (instead of $d - b - h$ positions).

$\therefore c \leqq \min\{d - b - h, b - (d_i - d)\}$

An Even Better Algorithm for Large Alphabets: idea



We randomly guess b and the b positions, and modify t at the b positions to agree with s_j;

further randomly guess h, the h positions, and the letters of t at the h positions, and modify t at the h positions accordingly.

Then, it remains to modify at most $\min\{d - b - h, b - (d_i - d)\}$ positions of t (instead of $d - b - h$ positions).

If σ is large, we are luckier than before.

An Even Better Algorithm for Large Alphabets: example

Example:

```
1:  GGGCGGCGTCA
2:  GATCGGCTGAG
3:  CACCGGCGTAG
4:  GAGCGGCGGGC
5:  GAGCCGCGAAG
6:  ACTCGGCGTAG
7:  GAAAGGCCTAG
8:  GAGCCGAGTGG
9:  GAGATGCGTTC
10: GACCGGGGTTG
```

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

$t =$ GACCGGGGTTG ← **The initial candidate solution.**

This t is not a solution yet.

9: GAGATGCGTTC ← **The input string farthest from t.**

$d_i = 5$

Among the d_i positions where the two strings disagree, we randomly guess b and the b positions, and modify t at the b positions to agree with s_j ; further randomly guess h, the h positions, and the letters of t at the h positions, and modify t at the h positions accordingly.

$b \geqq d_i - d$ $\quad$ $h \leqq d - b$

An Even Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGAGTTC ← **The candidate solution after 3 modifications.**

2: GATCGGCTGAG This t is not a solution yet. $d_i = 6$

However, we already modified d positions of t.
So, we give up this t and repeat the whole process from scratch.

An Even Better Algorithm for Large Alphabets: example

Example:

```
1:  GGGCGGCGTCA
2:  GATCGGCTGAG
3:  CACCGGCGTAG
4:  GAGCGGCGGGC
5:  GAGCCGCGAAG
6:  ACTCGGCGTAG
7:  GAAAGGCCTAG
8:  GAGCCGAGTGG
9:  GAGATGCGTTC
10: GACCGGGGTTG
```

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

$t =$ GACCGGGGTTG ← **The initial candidate solution.**

This t is not a solution yet.

9: GAGATGCGTTC ← **The input string farthest from t.**

$d_i = 5$

Among the d_i positions where the two strings disagree, we randomly guess b and the b positions, and modify t at the b positions to agree with s_j ; further randomly guess h, the h positions, and the letters of t at the h positions, and modify t at the h positions accordingly.

$b \geqq d_i - d$ $\quad$ $h \leqq d - b$

An Even Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

t = GAGCGGCGTTG ← **The candidate solution after 2 modifications.**

2: GATCGGCTGAG

This t is not a solution yet. $d_i = 4$

Among the positions where the two disagree, select $d_i - d$ positions uniformly at random, change the letter of t at each selected position to that of the other string with probability $\frac{2}{\sigma}$, while to each of the rest $\sigma - 2$ letters with probability $\frac{1}{\sigma}$.

An Even Better Algorithm for Large Alphabets: example

Example:

1: GGGCGGCGTCA
2: GATCGGCTGAG
3: CACCGGCGTAG
4: GAGCGGCGGGC
5: GAGCCGCGAAG
6: ACTCGGCGTAG
7: GAAAGGCCTAG
8: GAGCCGAGTGG
9: GAGATGCGTTC
10: GACCGGGGTTG

t = GAGCGGCGTAG ← **The candidate solution after 3 modifications.**

Idea: Try to modify at most d positions of an input string (say, the last input string) to obtain a solution.

$L = 11$
$d = 3$
$\sigma = 4$

This t is now a solution, because the Hamming distance between t and every input string is at most d.

An Even Better Algorithm for Large Alphabets: analysis

Lemma 5: If there is a solution s, then one repetition finds a solution with probability $= \Omega(d^{-1}\gamma^d)$, where $\varepsilon = \frac{d_1 - d}{d}$, d_1 is the largest Hamming distance between the initial t and an input string, and

- $\gamma = \max_{0 \leqq \alpha \leqq 1-\varepsilon} (\sigma-2)^{\alpha} \left(\frac{\sigma}{1-\varepsilon-\alpha}\right)^{\frac{1-\varepsilon-\alpha}{2}} \frac{(3+\varepsilon-\alpha)^{\frac{3+\varepsilon-\alpha}{2}}}{2^{\frac{1+\varepsilon-\alpha}{2}} \alpha^{\alpha} (1+\varepsilon-\alpha)^{1+\varepsilon-\alpha}}$

- $\sigma = 2$ (binary): $O(nL + nd^2 \cdot 5.83^d)$ expected time.
- $\sigma = 4$ (DNA): $O(nL + nd^2 \cdot 10.47^d)$ expected time.
- $\sigma = 20$ (protein): $O(nL + nd^2 \cdot 40.13^d)$ expected time.

slower than the second algorithm

slower than the 3-string algorithm

the fastest known algorithm

Combining with the second algorithm

Emulating the last and the second algorithms in parallel and halts once one of them stops yields the fastest algorithm.

Comparison of the bases of the powers in the time bounds (the exponent is *d*)

Algorithm	Reference	$\sigma = 2$	$\sigma = 4$	$\sigma = 20$	$\sigma = 50$	
3-String	[6]	6.73	13.18	51.23	113.3	old
CloseString2	[7]	8	13.92	47.21	100.4	
BoundedGuess	Theorem 4.2	5.44	10.87	54.37	135.9	new
NonRedundantGuess	Corollary 5.4	5	10	50	125	
LargeAlphabet1	Corollary 6.3	8	12	44	104	
LargeAlphabet2	Theorem 7.2	5.83	10.47	40.13	86.84	
NonRedundantGuess + LargeAlphabet2	Section 7	5	9.81	40.09	86.84	

Open problems

1. **Better analysis?**
2. **Even faster algorithms?**
3. **Derandomization?**