

# Reimplementation of the Shouji Sequence Pre-alignment filter

## Introduction

### The Problem!

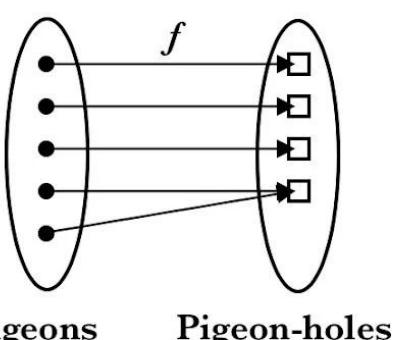
- Dynamic programming:  $\sim O(n^2)$
- Massive volume of sequence pairs/data in modern genomics
- Full alignment algorithms can waste time on obviously dissimilar pairs

→ Leads to very slow processing speeds 😞

### The Solution!💡

An algorithm that filters out all clearly dissimilar sequences before alignment!

Behold, **Shouji**! The pre-alignment filter based on the mathematical concept of the pigeonhole principle



"If you have more items than containers, at least one container must hold more than one item"

## Algorithm & Methods

$$\begin{aligned} n &= \text{sequence length} \\ E &= \text{edit threshold} \end{aligned}$$

Key Idea 🤝 :

If two sequences differ by  $E$  edits, they must share matching regions totaling  $\geq (n - E)$  characters

### Step 1: Build Neighborhood map

Text: G G T G C A G A G C T C  
Pattern: G G T G A G A G T T G T  
Match: ✓ ✓ ✓ ✓ X X ✓ ✓ X X X X  
Binary: 0 0 0 0 1 1 0 0 1 1 1 1

Shouji gets fed text and pattern sequences, then shifts the pattern left (+) or right (-)  $|E|$  times and re-compares with the text. These are called upper (+) and lower (-) diagonals

Each of the binary comparisons is put into a matrix (*our map!*) which evidently has  $2E+1$  rows

### Step 2: Sliding Window Search

Text: G G T G C A G A G C T C  
Pattern: G G T G A G A G T T G T

Check all 4-bit binary comparisons in the map that align with 4-bit window. Store the most matching (tie break rule = first 0 takes priority) in the *Shouji bit-vector*

Text: G [G T G C] A G A G C T C  
Pattern: G [G T G A] G A G T T G T

Slide window over and find SBV. Keep sliding until completed.

### Step 3: Count and Decide

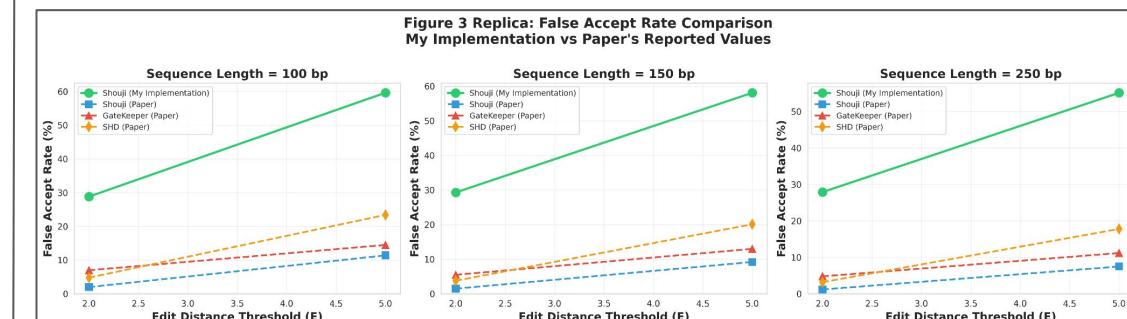
Count total mismatches (1's) in Shouji bit-vector. If the total number of 1's is greater than the edit threshold ( $E$ ), then the sequence pair is rejected.

If not, the sequence pair can be passed on to the alignment tool.

## Testing and Results

I was able to successfully implement this algorithm myself, testing on random data I generated.

The main form of quality testing done was analyzing the algorithms FAR and FRR. In this, my implementation had some imperfections when compared to the true Shouji Implementation. My version shows a way higher FAR (28-60%) than what the true version shows (2-15%), as shown below.



My version, though, was stronger in its FRR, with an average of 0.516% across all tests. Though Magnet also shared a >0% FRR, the true Shouji implementation, though, claims to provide a 0% FRR rate.

Table 2 Replica: Filtering Accuracy Results  
My Shouji Implementation Performance Metrics

Config	Seq Length	E Thresh	Total Pairs	Similar	Dissimilar	FAR (%)	FRR (%)	Accuracy (%)
L100_E2	100	2	5000	2374	2626	28.83	1.18	84.30
L100_E5	100	5	5000	2328	2672	59.62	0.04	68.12
L150_E2	150	2	5000	2395	2605	29.29	0.84	84.34
L150_E5	150	5	5000	2234	2766	58.06	0.04	67.86
L250_E2	250	2	5000	2327	2673	27.91	0.95	84.64
L250_E5	250	5	5000	2188	2812	55.12	0.05	68.98

## Conclusions 🤝

- Core algorithm successfully implemented
- Higher FAR (28-60% vs. paper's 2-15%) reflects synthetic data and conservative tuning. This tradeoff is acceptable.
- Successful algorithms require appropriate platforms. While pure Python validates correctness, production performance demands hardware acceleration → FPGA chip gives ~1000x speedup