ALGORITHMS OF BIOINFORMATICS

5

String Matching

4 December 2025

Prof. Dr. Sebastian Wild

CS594 (Winter 2025-26)
Philipps-Universität Marburg
version 2025-12-12 11:39 H

Outline

5 String Matching

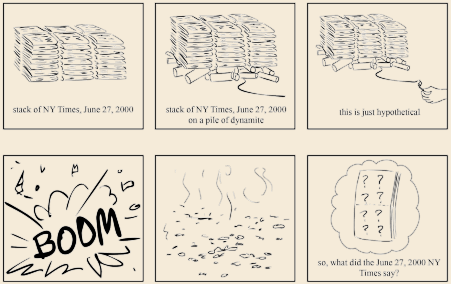
5.1 Read Mapping
5.2 String Matching with Finite Automata
5.3 Constructing String Matching Automata
5.4 The Knuth-Morris-Pratt algorithm
5.5 The Aho-Corasick Algorithm
5.6 Alignment-Based Matching
5.7 Longest Common Extensions
5.8 SNPs via filtering

5.1 Read Mapping

Shotgun Sequencing

Recall:

- *Shotgun sequencing* approach to determine a (human) genome:



Compeau & Pevzner, *Bioinformatics Algorithms*, Fig 3.1
https://cogniterra.org/lesson/29884/step/2?unit=21982

- For a single human genome need 300M reads of 200bp (30x coverage)
  - ⇝ 60 GB of raw data

⇝ genome assembly from those is expensive and error prone

We now have carefully assembled reference genomes to compare with!

Shotgun Sequencing in Medicine

Predominant medical use of whole genome sequencing:
detecting known markers (mutations / gene combinations) for diseases

Example: (more details in Compeau & Pevzner 2015)

- *Ohdo syndrome* (form of mental retardation, "mask-like" face)
- known to be indicated by single protein-truncating mutation

Often even a **single base replacement** w.r.t. a human reference genome,
a ***SNP (single nucleotide polymorphism)***

⇝ *To find SNPs, we don't need a new patient's genome fully assembled!*

Read Mapping

- We thus work towards solving the ***read mapping problem***
  - **Given:** genome/text $T[0..n)$, reads/patterns $P[0..p)$, $P[r] = P_r[0..m_r)$
  - **Goal:** locations i_r of *"best match"* for P_r in T for $r \in [0..p)$.
- "best match" can be interpreted in several ways, leading to different problems:
  (a) best **semilocal alignment** of P_r to T (gold standard, usually too expensive)
  (b) match with **fewest mismatches**
  (c) match with $\leq d$ **mismatches** or `NO_MATCH` if no such exists
  for SNPs can even set $d = 1$
  (d) **exact match** or `NO_MATCH` if no such exists

*We will first focus on **exact matches**.*

- simplifies the problem (to get started)
- the SNPs variants can be reduced to it (using postprocessing)

Part I

Exact matches

Notation

- *alphabet* Σ: finite set of allowed **characters**; $\sigma = |\Sigma|$ *"a string over alphabet Σ"*
  - focus on nucleotides $\{A, C, G, T\}$ and amino acids
  - but try to keep methods generic
  - letters (Latin, Greek, Arabic, Cyrillic, Asian scripts, ...) Unicode characters
    comprehensive standard character set including emoji and all known symbols
- $\Sigma^n = \Sigma \times \cdots \times \Sigma$: strings of **length** $n \in \mathbb{N}_0$ (n-tuples)
  - $\Sigma^\star = \bigcup_{n \geq 0} \Sigma^n$: set of **all** (finite) strings over Σ, $\Sigma^+ = \bigcup_{n \geq 1} \Sigma^n$
  - $\varepsilon \in \Sigma^0$: the *empty* string (same for all alphabets)
- for $S \in \Sigma^n$, write $S = S[0..n)$, so $S[i]$ (other sources: S_i) for ***i*th** character $(0 \leq i < n)$
  zero-based (like arrays)!
- for $S, T \in \Sigma^\star$, write $ST = S \cdot T$ for **concatenation** of S and T
- for $S \in \Sigma^n$, write $S[i..j)$ for the **substring** $S[i] \cdot S[i+1] \cdots S[j-1]$ $(0 \leq i \leq j \leq n)$
  - $S[i..i) = \varepsilon$
  - $S[0..j)$ is a **prefix** of S; $S[i..n)$ is a **suffix** of S

String matching – Definition

Search for a string (pattern) in a large body of text

- **Input:**
  - $T \in \Sigma^n$: The *text* being searched within
  - $P \in \Sigma^m$: The *pattern* being searched for; typically $n \gg m$
- **Output:**
  - the *first occurrence (match)* of P in T: $\min\{i \in [0..n-m) : T[i..i+m) = P\}$
  - or `NO_MATCH` if there is no such i ("P does not occur in T")
  - sometimes also: find **all** occurrences of P in T.

try all starting positions

- trivially solvable with $(n-m+1) \cdot m \sim nm$ character comparisons

$\rightsquigarrow$ too slow for read mapping!

- string matching available, e. g., Java in `String.indexOf`, Python in `str.find`
  - not always robust enough for bioinformatics data
    (small alphabet, long repetitions)

5.2 String Matching with Finite Automata

Theoretical Computer Science to the rescue!

- string matching = deciding whether $T \in \Sigma^\star \cdot P \cdot \Sigma^\star$
- $\Sigma^\star \cdot P \cdot \Sigma^\star$ is *regular* formal language

$\rightsquigarrow$ $\exists$ *deterministic finite automaton* (DFA) to recognize $\Sigma^\star \cdot P \cdot \Sigma^\star$

$\rightsquigarrow$ can check for occurrence of P in $|T| = n$ steps!

Job done!

WTF!?

We are not quite done yet.

- (Problem 0: programmer might not know automata and formal languages . . .)
- Problem 1: existence alone does not give an algorithm!
- Problem 2: automaton could be very big!

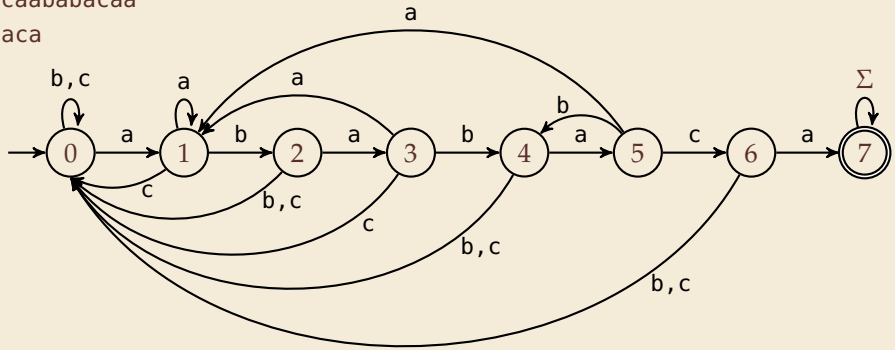
String matching with DFA

- Assume first, we already have a deterministic automaton
- How does string matching work?

Example:

T = aabacaababacaa

P = ababaca



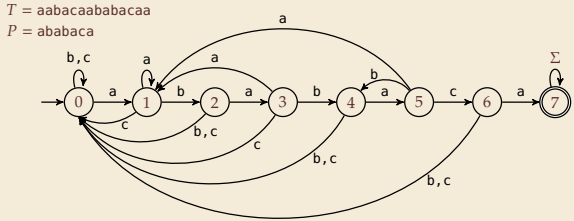
text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

String matching DFA – Intuition

Why does this work?

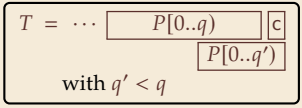
- Main insight:

> State q means:
> *"we have seen $P[0..q)$ until here (but not any longer prefix of P)"*



text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	1	1	2	3	0	1	1	2	3	4	5	6	7	7

- If the next text character c does not match, we know:
  - (i) text seen so far ends with $P[0...q) \cdot c$
  - (ii) $P[0...q) \cdot c$ is not a prefix of P
  - (iii) without reading c, $P[0..q)$ was the *longest* prefix of P that ends here.



⇝ New longest matched prefix will be (weakly) shorter than q

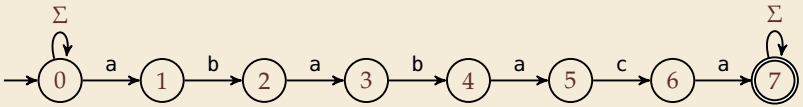
⇝ All information about the text needed to determine it is contained in $P[0...q) \cdot c$!

5.3 Constructing String Matching Automata

NFA instead of DFA?

It remains to *construct* the DFA.

- trivial part:



- that actually is a ***non****deterministic finite automaton* (NFA) for $\Sigma^\star P\, \Sigma^\star$

⇝ We *could* use the NFA directly for string matching:

- at any point in time, we are in a ***set*** **of states**
- accept when one of them is final state

Example:

text:		a	a	b	a	c	a	a	b	a	b	a	c	a	a
state:	0	0, 1	0, 1	0, 2	0, 1, 3	0	0, 1	0, 1	0, 2	0, 1, 3	0, 2, 4	0, 1, 3, 5	0, 6	0, 1, 7	0, 1, 7

But maintaining a whole set makes this slow . . .

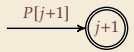
Computing DFA directly

You have an NFA and want a DFA?
Simply apply the power-set construction
(and maybe DFA minimization)!

```
The powerset method has exponential state blow up!
I guess I might as well use brute force string matching ...
```

Ingenious algorithm by Knuth, Morris, and Pratt: construct DFA *inductively*:

Suppose we add character $P[j]$ to automaton A_j for $P[0..j)$ to construct A_{j+1}

- add new state and matching transition $\rightsquigarrow$ easy 

- for each $c \neq P[j]$, we need $\delta(j, c)$ (transition from j when reading c)
- $\delta(j, c)$ = length of the longest prefix of $P[0..j)c$ that is a suffix of $P[1..j)c$
  = state of automaton after reading $P[1..j)c$
  $\leq$ j $\rightsquigarrow$ can use known automaton A_j for that!

State q means:
"we have seen $P[0..q)$ until here (but not any longer prefix of P)"

$\rightsquigarrow$ can directly compute A_{j+1} from A_j!


seems to require simulating automata $m \cdot \sigma$ times

Computing DFA efficiently

- **KMP's second insight:** simulations in one step differ only in last symbol

$\rightsquigarrow$ simply maintain state x, the state after reading $P[1..j)$.

- copy its transitions
- update x by following transitions for $P[j]$

```
1  procedure constructDFA(P[0..m)):
2      // δ[q][c] = target state when reading c in state q
3      for c ∈ Σ do
4          δ[0][c] := 0
5      δ[0][P[0]] := 1
6      x := 0
7      for j = 1, ..., m − 1 do
8          for c ∈ Σ do // copy transitions
9              δ[j][c] := δ[x][c]
10         δ[j][P[j]] := j + 1 // match edge
11         x := δ[x][P[j]] // update x
```

Example: $P[0..7) =$ `ababaca`

$\delta(c, q)$	0	1	2	3	4	5	6
a	1	1	3	1	5	1	7
b	0	2	0	4	0	4	0
c	0	0	0	0	0	6	0

String matching with DFA – Discussion

- **Time:**
  - Matching: n table lookups for DFA transitions
  - building DFA: $\Theta(m\sigma)$ time (constant time per transition edge).

  $\rightsquigarrow$ $\Theta(m\sigma + n)$ time for string matching.

- **Space:**
  - $\Theta(m\sigma)$ space for transition matrix.


fast matching time actually: hard to beat!


total time asymptotically optimal for small alphabet (for $\sigma = O(n/m)$)


substantial **space overhead**, in particular for large alphabets

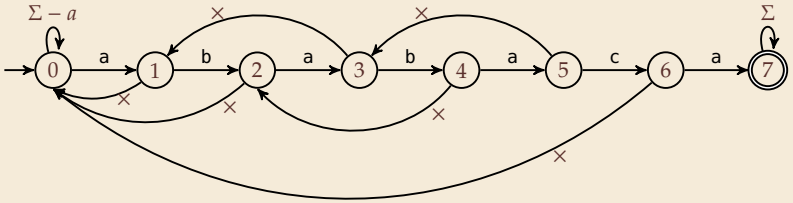
5.4 The Knuth-Morris-Pratt algorithm

Failure Links

- Recall: String matching with DFA is fast,
  but needs table of $m \times \sigma$ transitions.
- in fast DFA construction, we used that all simulations differ only by *last* symbol

$\rightsquigarrow$ **KMP's third insight:** do this last step of simulation from state x during *matching*!
. . . but how?

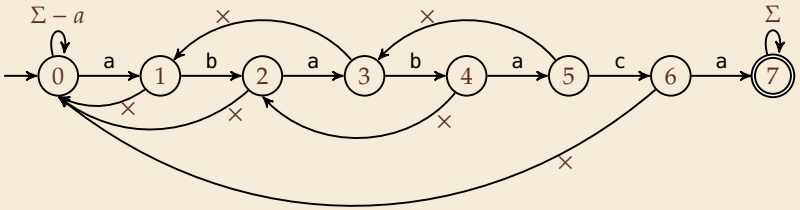
- **Answer:** Use a new type of transition: $\times$, the *failure links*
  - Use this transition (only) if no other one fits.
  - $\times$ *does not consume a character.* $\rightsquigarrow$ might follow several failure links



$\rightsquigarrow$ Computations are deterministic (but automaton is not a classic DFA.)

Failure link automaton – Example

Example: T = abababaaaca, P = ababaca



T :	a	b	a	b	a	b	a	a	b	a	b
P :	a	b	a	b	a	×					
			(a)	(b)	(a)	b	a	×			
								a	b	a	b

to state 3

to state 1

q :	1	2	3	4	5	3, 4	5	3, 1, 0, 1	2	3	4

(after reading this character)

The Knuth-Morris-Pratt Algorithm

```
1  procedure KMP(T[0..n), P[0..m)):
2      fail[0..m] := failureLinks(P)
3      i := 0 // current position in T
4      q := 0 // current state of KMP automaton
5      while i < n do
6          if T[i] == P[q] then
7              i := i + 1; q := q + 1
8              if q == m then
9                  return i − q // occurrence found
10         else // i.e. T[i] ≠ P[q]
11             if q ≥ 1 then
12                 q := fail[q] // follow one ×
13             else
14                 i := i + 1
15     end while
16     return NO_MATCH
```

- only need single array *fail* for failure links
- (failureLinks on next slide)

Analysis: (matching part)

- always have $fail[j] < j$ for $j \geq 1$

$\rightsquigarrow$ in each iteration

- either advance position in text ($i := i + 1$)
- or shift pattern forward (guess $i - q$)

- each can happen at most n times

$\rightsquigarrow \leq 2n$ symbol comparisons!

Computing failure links

- failure links point to error state x (from DFA construction)

$\rightsquigarrow$ run same algorithm, but store $fail[j] := x$ instead of copying all transitions

```
1  procedure failureLinks(P[0..m)):
2      fail[0] := 0
3      x := 0
4      for j := 1, ..., m − 1 do
5          fail[j] := x
6          // update failure state using failure links:
7          while P[x] ≠ P[j]
8              if x == 0 then
9                  x := −1;  break
10             else
11                 x := fail[x]
12         end while
13         x := x + 1
14     end for
```

Analysis:

- m iterations of for loop
- while loop always decrements x
- x is incremented only once per iteration of for loop

$\rightsquigarrow \leq m$ iterations of while loop *in total*

$\rightsquigarrow \leq 2m$ symbol comparisons

Knuth-Morris-Pratt – Discussion

- **Time:**
  - $\leq 2n + 2m = O(n + m)$ character comparisons
  - clearly must at least *read* both T and P in the worst case
  - $\rightsquigarrow$ KMP has optimal worst-case complexity

- **Space:**
  - $\Theta(m)$ space for failure links


total time asymptotically optimal (for any alphabet size)


reasonable extra space

The KMP prefix function

- It turns out that the failure links are useful beyond KMP
- a slight variation is (more?) widely used: (for historic reasons)
  the (KMP) *prefix function* $F : [1..m-1] \to [0..m-1]$:

  $F[j]$ ***is the length of the longest prefix of*** $P[0..j]$
  that is a suffix of $P[1..j]$**.**

- Can show: $fail[j] = F[j-1]$ for $j \geq 1$, and hence

  $fail[q]$ = ***length*** *of the longest prefix of* $P[0..q)$ *that is a suffix of* $P[1..q)$.

  ← memorize this!

- EAA Buch: String indices are 1-based, but definition of failure links matches! $\boxed{\Pi_P(q) = fail[q]}$

  $\Pi_P : [1..m] \to [0..m-1]$ with $\Pi_P(q) = \max\{k \in \mathbb{N}_0 : k < q \wedge P[0..k) \sqsupset P[0..q)\} = fail[q]$

5.5 The Aho-Corasick Algorithm

Multiple-Pattern Matching

- So far: process a single pattern P in one pass over T.

$\rightsquigarrow$ Cannot beat time $\Omega(p \cdot n)$

- The essence of KMP can be generalized to deal with several patterns!

$\rightsquigarrow$ The ***Aho-Corasick Algorithm***

The Multiple-Pattern Matching Problem

- **Given:** text $T[0..n)$, patterns $P[0..p)$, $P[r] = P_r[0..m_r)$
  - all over $\Sigma = [0..\sigma)$ for **constant** σ
  - total length of patterns: $m := \sum_{0 \le r < p} m_r$
- **Goal:** all matches, i. e., all pairs (i, r) such that $P_r = T[i..i + m_r)$

single pass!

Aho-Corasick can do this with $O(m)$ preprocessing and $O(n + output)$ matching time!
Here $output$ is the number of match pairs (i, r).

Aho-Corasick Automaton – Overview

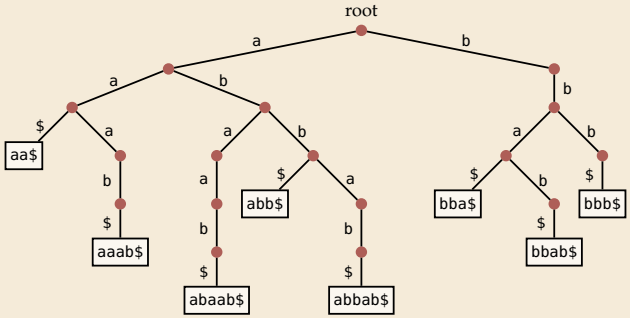
Aho-Corasick Automaton

1. Build trie A from patterns $P[0..p)$.
2. Add *failure links* to A.
3. Add *output links* to A.

Recap: Tries

- efficient dictionary data structure for strings
- name from re**trie**val, but pronounced "try"
- tree based on symbol comparisons

- **Example**:
  {aa$, aaab$, abaab$, abb$, abbab$, bba$, bbab$, bbb$}



When stored string is a strict prefix of another, internal nodes can correspond to strings.

Aho-Corasick Automaton – Adding Failure Links

Trie for P_r corresponds to match-edges-only NFA.

$\rightsquigarrow$ Interpreting the trie as automaton, add ε-edges back to the root.

- as in KMP, instead of determinizing the automation classically, we again use failure links

$\rightsquigarrow$ construction as for KMP using failure state x, repeated for each word.

Aho-Corasick Automaton – Output Links

An automaton state might contain other patterns as suffix $\rightsquigarrow$ *must output match(es)! But we are not in an accepting state, so direct use of automaton so far would miss occurrence!*

- ***output links:*** each state points to longest suffix pattern (if any).
  - During matching, traverse linked list of matches and output each $\rightsquigarrow$ $O(output)$ cost overall
  - Computation of output links similar to failure links! (handle patterns by length)

Aho-Corasick Summary

- Both failure links and output links can be computed in BFS of pattern trie

$\rightsquigarrow$ total construction time $O(m)$

- for read mapping, split reads into batches, so Aho-Corasick automaton of batch fits in memory

$\rightsquigarrow$ e. g., 300 batches of 1M reads for human genome

$\rightsquigarrow$ substantial speedup over reading the reference genome millions of times!

Part II

Inexact Matches

Robust Mapping

- so far: only able to find reads that *perfectly* match reference genome

$\rightsquigarrow$ biologically insufficient

- For the moment, focus again on a single pattern $P[0..m)$

5.6 Alignment-Based Matching

Fitting Alignment

- **Gold Standard:** best *alignment score* over all substrings of T with P.

⇝ Can find this as a pairwise alignment! ⇝ Recall Unit 3

Semilocal Aligment a.k.a. Fitting Alignment

Slight twist: We know conserved region, but need to find best match in larger sequence.
What substring of $B[0..n)$ is the best match for $A[0..m)$? (typically then $m \ll n$)

⇝ only allow IgnorePrefix($B[0..j)$) and IgnoreSuffix($B[j..n)$)

$$\rightsquigarrow\ D(i,j) = \begin{cases} -i & \text{if } j = 0 \\ \mathbf{0} & \text{if } i = 0 \\ \max \begin{cases} D(i-1,j) \mathbf{-1}, \\ D(i,j-1) \mathbf{-1}, \\ D(i-1,j-1) + \left[A[i-1] \mathbf{=} B[j-1]\right] \end{cases} & \text{otherwise} \end{cases}$$

Optimal local alignment score: $\max_{j \in [0..n]} D[m][j]$

13

⇝ naive time and space complexity $\Theta(n \cdot m)$ (ignoring exhaustive tabulation)

Slightly more space efficient fitting alignments

Idea: don't rely on minimization over i' at end

- $D(i,j)$ = score of best alignment of $T[i'..i)$ to $P[0..j)$ for any $i' \in [0..i]$

$$D(i,j) = \begin{cases} \mathbf{0} & \text{if } j = 0 \\ j \cdot g & \text{if } i = 0 \\ D(i-1,j) & \text{if } j = m \\ \min \begin{cases} D(i-1,j) + g \cdot \mathbf{[j < m]}, \\ D(i,j-1) + g, \\ D(i-1,j-1) + p(A[i-1], B[j-1]) \end{cases} & \text{otherwise} \end{cases}$$

$\rightsquigarrow$ same time complexity $\Theta(n \cdot m)$ (ignoring exhaustive tabulation)

- but better space usage
  - suffices to last 2 columns $\rightsquigarrow$ $\Theta(m)$ space
  - can remember i' in first case $\rightsquigarrow$ no need to reconstruct alignment for match

👎 still usually prohibitively slow

👎 even limiting to edit distance $\leq k$ (banded alignment) doesn't speed up DP

5.7 Longest Common Extensions

k-Mismatch Matching

Let's first simplify the problem a bit.

- instead of a match of P with minimal **edit** distance to a substring of T
  find a match of P with minimal **Hamming** distance to a substring of T

$\rightsquigarrow$ only length-m substrings of $T[i..i+m)$ relevant, $i \in [0..n-m)$

How fast can we solve this problem?

OPEN!

- survey of techniques

Navarro: *A Guided Tour to Approximate String Matching*, ACM Comp. Surv. **2001**

- recent work at FOCS 2020, FOCS 2023

Hamming distance – Brute Force

- We can easily adapt the brute-force string matching to
  - find all matches with $\leq k$-mismatches
  - find a match with fewest mismatches

```
1 procedure kMismatchFind(T[0..n), P[0..m), k):
2     for i := 0, ..., n − m − 1 do
3         mismatches := 0
4         for j := 0, ..., m − 1 do
5             if T[i + j] ≠ P[j] then mismatches := mismatches + 1
6             if mismatches > k then break inner loop
7         if j == m then return i
8     return NO_MATCH
```


Simple, easy to parallelize


No extra space, no preprocessing


time $\Theta(nm)$ in the worst case
even when $k \ll m$

Longest Common Extensions

- To do better, we need a way to quickly skip matching parts.

  Enter: ***longest common extension (LCE)*** data structure:

  - **Given:** String $T[0..n)$
  - **Goal:** Answer LCE queries, i. e.,
    given positions i, j in T,
    how far can we read the same text from there?
    formally: $\mathrm{LCE}(i,j) = \max\{\ell : T[i..i+\ell) = T[j..j+\ell)\}$

Amazing result: Can compute data structure in $\Theta(n)$ time and space that finds any LCE in **constant(!) time**.

- we will ultimately see the full result in the coming units

$\rightsquigarrow$ for now, use $O(1)$ LCA as black box.

$\rightsquigarrow$ After linear preprocessing (time & space), we can find LCEs in $O(1)$ time.

Kangaroo Algorithm for approximate matching

- Note: LCE defined for single string; we will need it for i in T and j in P

$\rightsquigarrow$ build LCE data structure on $T' = T\$_1 P\$_2$ in time $O(n+m)$

```
1  procedure kMismatch(T[0..n − 1], P[0..m − 1]):
2      // build LCE data structure
3      for i := 0, ..., n − m − 1 do
4          mismatches := 0;  t := i;  p := 0
5          while mismatches ≤ k ∧ p < m do
6              ℓ := LCE_{T'}(t, p) // jump over matching part
7              t := t + ℓ + 1;  p := p + ℓ + 1
8              mismatches := mismatches + 1
9          if p == m then
10             return i
```

- **Analysis:** $\Theta(n+m)$ preprocessing + $O(n \cdot k)$ matching

very efficient for small k for a **single pattern**

Kangaroo Algorithm – Discussion

- as given above, uses $\Omega(n)$ preprocessing **per pattern**,
  but can built joint LCE data structure for several patterns

👎 (with blackbox reuse), preprocessing on T must be redone for each set of patterns

👎 working space $\Theta(n + m)$

5.8 SNPs via filtering

From Exact to Inexact

- Overhead of LCE data structure can be sizable.
- For very small k, a much simpler heuristic can be effective
- **Observation:** If a pattern $P[0..m)$ matches a text T with $\leq k$ mismatches, there is an **exact** match of a substring of length $\geq t := \left\lfloor \frac{m}{k+1} \right\rfloor$.

$\rightsquigarrow$ Break P into $k+1$ substrings $P'_1, \ldots, P'_{k+1}$ of length t (except potentially longer last substring)

- Find *"seeds"*: exact occurrences of $P'_1, \ldots, P'_{k+1}$ (e. g., using Aho-Corasick)
- Try to extend each seed to a full $\leq k$-mismatch match

👎 Worst case: find $\Theta(nk)$ matches of a t-substring of P, but no k-mismatches occurrence

$\rightsquigarrow$ Time to check all spurious matches $\Omega(nm)$

👍 For $m \gg k$ and biological sequences, spurious matches are rare

👍 seed-extension can be generalized to compute best **alignment** around a seed

Practical Heuristics

- many practical bioinformatics tools make use of seed extension
- FASTA and BLAST are heuristic methods for optimal local alignments
  - no approximation guarantee for returned alignments
  - BLAST widely used in practice, e. g., for GenBank searches
  - biological goal: find unknown connections, e. g., *homologous sequences*
- BLAST starts with short seeds with few mismatches
  - for a k-mer P and alignment score threshold σ, compute *all* strings P' with $dist(P, P) \leq \sigma$
  - search for exact matches of all these P'

Summary

- **exact** matching: $O(n + m + output)$ for single or multiple patterns
- k-**mismatch** matching:
  - $O(nk)$ after $O(n + m)$ preprocessing
  - (exact) seed extension often linear time
- k-**distance** matching:
  - gold standard semilocal alignment $O(nm)$
  - improvement in tutorials
  - (heuristic) seed extension methods often linear time, often "precise enough"