

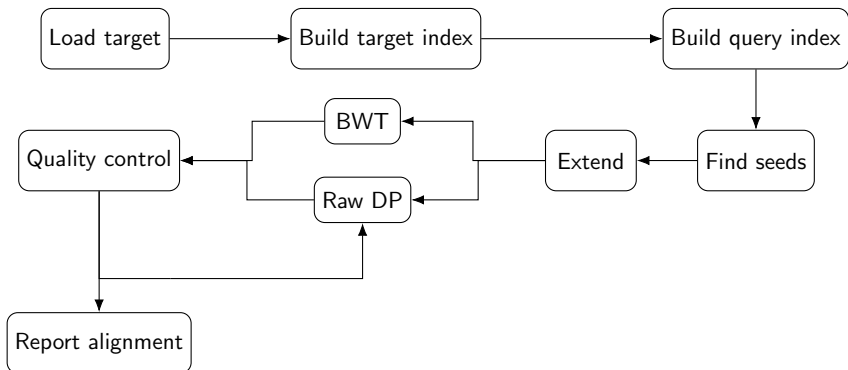
# Fast Python sequence aligner

Piotr Styczyński

MIM UW

2024

# Seed and Extend approach



# Aligner algorithm description

1. Build reference index structures
  - 1.1 Read reference in chunks of size 800000
  - 1.2 For each chunk generate (k=17, w=8)-minimizer index chunk
  - 1.3 Filter out every other 15-th k-mer (only if it occurs only once within the chunk)
  - 1.4 Merge chunk index into full reference index and repeat for other chunks
2. Load query and build (k=17, w=8)-minimizer index
3. Find common matches and generate cross-product
4. Perform seed extension (described in detail later) and generate matching positions with scores
5. Filter 5 top scores  $\geq 0.1$ 
  - 5.1 For  $|scores| = 0$  try building full LIS and use sliding window with highest scoring:  $\min(|match_q|, |match_t|)^2 - \sum diff(match_t)$
  - 5.2 For  $|scores| = 1$  proceed normally
  - 5.3 For  $|scores| > 1$  get region with lowest score generated by DP aligner
6. Perform final one region matching

# Aligner algorithm description

6. Perform final one region matching
  - 6.1 For every region virtually resize it by  $kmer\_len * 0.5 + 1 = 9$  each side
  - 6.2 For every region do quick BWT backtrack alignment on 10% of match (start and end) with max err. rate  $10\% = 10$
  - 6.3 If BWT finds match on end but padding exceeds 4% of match length, execute DP aligner (but only on first parameter configuration i.e (15, 11))
    - 6.3.1 DP aligner takes prefix/suffix of 40% of query size
    - 6.3.2 For  $(kmer\_len, step) = ((15, 11), (8, 5))$  it tries to match the pair target/sequence
    - 6.3.3 Run recursive algorithm with threshold of errors  
 $\leq (|query| * 11.11\%)$
  - 6.4 Now if padding returned by BWT or/and DP aligners make region exceed size of  $|query| * 105\%$  then repeat BWT matching process with new estimate  $start = found\_end - |query|$
  - 6.5 If we repeated procedure of running BWT/DP twice and came here again, then we assume there is no good match
7. Filter out matches if region exceed size of  $|query| * 105\%$
8. Print matches coordinates

# Problem of extending seeds

- ▶ minimap2 approach with dynamic programming similar to normal alignment (plus exponential forward lookups)
- ▶ We have only one potential match so maybe assume that  $match \in LIS(matches)$
- ▶ Can we formulate sliding window approach to generate the biggest sporing windows included in  $LIS(matches)$ ?
- ▶ For which case assumption  $match \in LIS(matches)$  won't work?
- ▶ Can we modify LIS to perhaps include other potential matches?

# Seed extension

---

**Algorithm** Standard LIS construction  $O(n \log n)$

---

**Require:**  $n \geq 0$

$lis\_len \leftarrow 0$

▷Length of LIS

$parent \leftarrow \{\infty, \infty, \infty, \dots, \infty\}_{n+1}$

▷Mapping to reconstruct LIS

$sub \leftarrow \{\infty, \infty, \infty, \dots, \infty\}_{n+1}$

▷Array with indices for matches that form LIS

$i \leftarrow 0$

**while**  $i < n$  **do**

▷Iterate over all elements  $i = 0, 1, 2, \dots, n - 1$

$start \leftarrow 1$

$end \leftarrow lis\_len$

**while**  $start \leq end$  **do**

    ▷Binary search over existing longest sequence

$middle \leftarrow \left\lfloor \frac{start+end}{2} \right\rfloor$

**if**  $matches_q[sub[middle]] < matches_q[i]$  **then**

$start \leftarrow middle + 1$

**else**

$start \leftarrow middle - 1$

$parent[i] \leftarrow sub[start - 1]$

▷We pin current value to the found parent

$sub[start] \leftarrow i$

**if**  $start > lis\_len$  **then**

$lis\_len = start$

$i \leftarrow i + 1$

---

## Seed extension

---

**Algorithm** Reconstruct LIS by following parent array  $O(n)$

---

$current\_node \leftarrow sub[lis\_len]$

$result \leftarrow \{0, 0, 0, \dots, 0\}_{lis\_len}$

$result[lis\_len - 1] \leftarrow current\_node$   $\triangleright$  Will contain all indices from matches describing the output subsequence

$j \leftarrow lis\_len - 1$

**while**  $1 \leq j$  **do**

$current\_node \leftarrow parent[current\_node]$

$result[j - 1] \leftarrow current\_node$

$j \leftarrow j - 1$

---

# Seed extension

---

## Algorithm Segmented-LIS heuristic $O(n \log n)$

---

Require:  $n \geq 0$

$lis\_len \leftarrow 0$

▷ Length of LIS

$parent \leftarrow \{\infty, \infty, \infty, \dots, \infty\}_{n+1}$

▷ Mapping to reconstruct LIS

$sub \leftarrow \{\infty, \infty, \infty, \dots, \infty\}_{n+1}$

▷ Array with indices for matches that form LIS

$i \leftarrow 0$

**while**  $i < n$  **do**

▷ Iterate over all elements  $i = 0, 1, 2, \dots, n - 1$

$start \leftarrow 1$

$end \leftarrow lis\_len$

**while**  $start \leq end$  **do**

▷ Binary search-like

$middle \leftarrow \lfloor \frac{start+end}{2} \rfloor$

**if**  $matches_T[sub[middle]] > matches_T[i] - max\_diff$  **then**

▷ Encountered old entry

$end \leftarrow start - 1$

▷ Breaks loop

**else if**  $matches_Q[sub[middle]] < matches_Q[i]$  **then**

$start \leftarrow middle + 1$

**else**

$start \leftarrow middle - 1$

$parent[i] \leftarrow sub[start - 1]$

▷ We pin current value to the found parent

$sub[start] \leftarrow i$

**if**  $start > lis\_len$  **then**

$lis\_len = start$

$i \leftarrow i + 1$

---



## Why does it seems to be working?

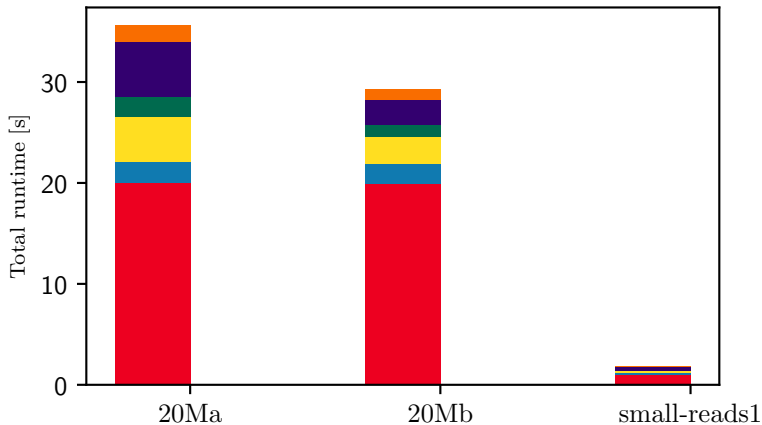
- ▶ The case when normal LIS does not work is first worse match with long tail of occurrences beforehand
- ▶ Finishing binary-search early on the left side makes us reuse previous sequences, the more right selections we made
- ▶ So for each turn right our probability increases
- ▶ Hence for 3 good matches we will have a high chance of at least having its part in the final array
- ▶ Of course with increasing number of candidates (especially with lower starting positions) it will work worse and worse

## What after we generated LIS heuristic?

- ▶ We can do the sliding window technique as query indices will be monotonic (locally) after we encounter target gap
- ▶ Hence we can formulate simple algorithm for each starting position  $i$  in our heuristical-LIS array
  1. For each  $i$  find last available target position  $j$  (using binary search)
  2. For window we define  $spaces(s) = \sum_{\substack{k \in \{1,2,\dots,f\}, \\ match[k] \in window[i,j]}} \left( \frac{\min(match[k]_s - match[k-1]_s, kmer\_len)}{kmer\_len} \right)$
  3. Calculate window score using:
$$score \leftarrow \frac{\min(|match_q|, |match_t|) - \max(spaces_t, spaces_q)}{|query|}$$
  4. If the score is local maximum i.e  $score_{i-1} < score_i \wedge (score_i > score_{i+1} \vee i+1 == |LIS|)$  then add it to the max scores bucket
$$max\_scores[\frac{i}{|query|*10\%}] \leftarrow max\_scores[\frac{i}{|query|*10\%}] \cup \{score_i\}$$
- ▶ Filter 5 top scores  $\geq 0.1$  (see previous slides)

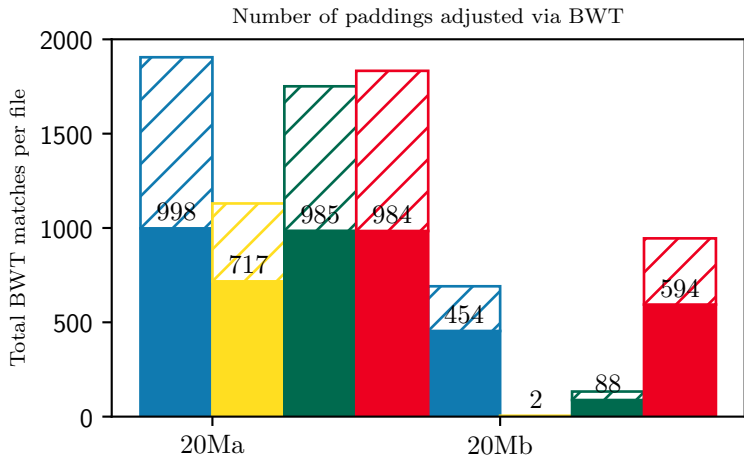
# Execution times

Total runtime by category

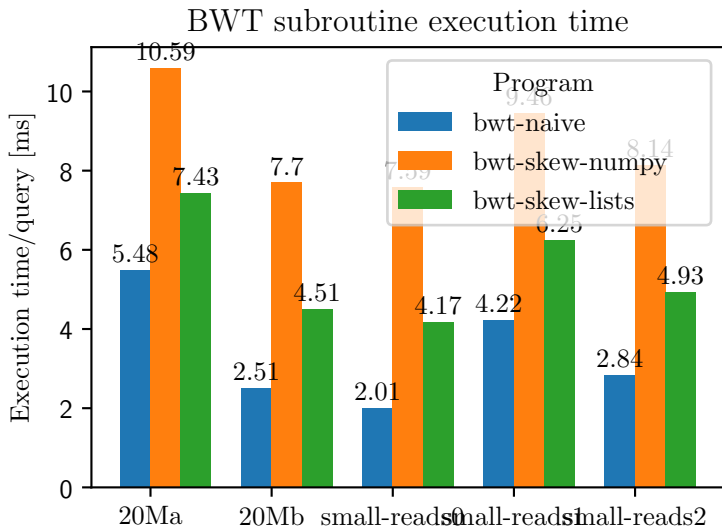


- solution: Index target
- solution: Index query
- solution: Read LIS
- solution: Find best LIS region
- solution: BWT Alignment
- solution: DP Alignment

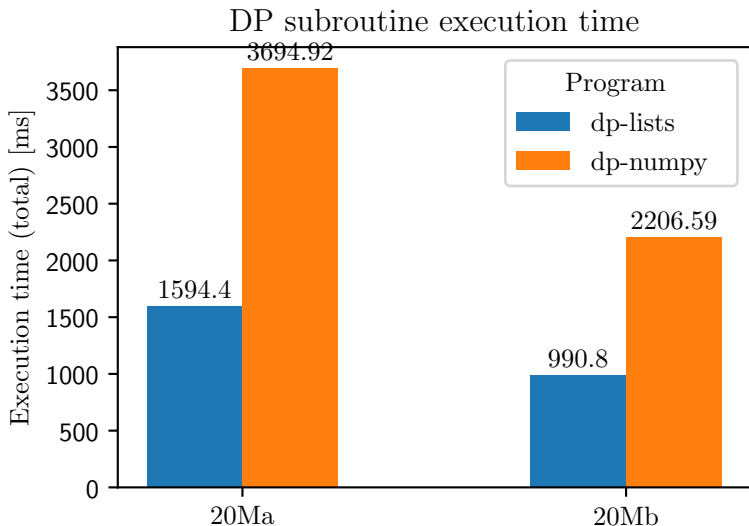
# Aligner routine effectiveness



# BWT routine implementation



# Raw DP routine implementation



## Match quality evaluation

Assuming we have match  $(r_1, r_2)$  and expect  $(e_1, e_2)$  We define the following:

- ▶  $d_1 := |e_1 - r_1|$
- ▶  $d_2 := |e_2 - r_2|$
- ▶  $d_s := d_1 + d_2$
- ▶  $d_m := \max(d_1, d_2)$
- ▶  $\text{score}((r_1, r_2)) = AA$  iff  $d_s < 10$
- ▶  $\text{score}((r_1, r_2)) = AB$  iff  $d_s < 20$
- ▶  $\text{score}((r_1, r_2)) = C$  iff  $d_m < 20$
- ▶  $\text{score}((r_1, r_2)) = D$  iff  $d_m \geq 20$

case name	ok	unmapped	AA	AB	C	D (bad)
reads20Mb	1000	0	936	64	0	0
reads20Ma	1000	0	975	25	0	0
reads2	100	0	90	10	0	0

*Thank you*