DS 202: Project Presentation

on

Minimizers

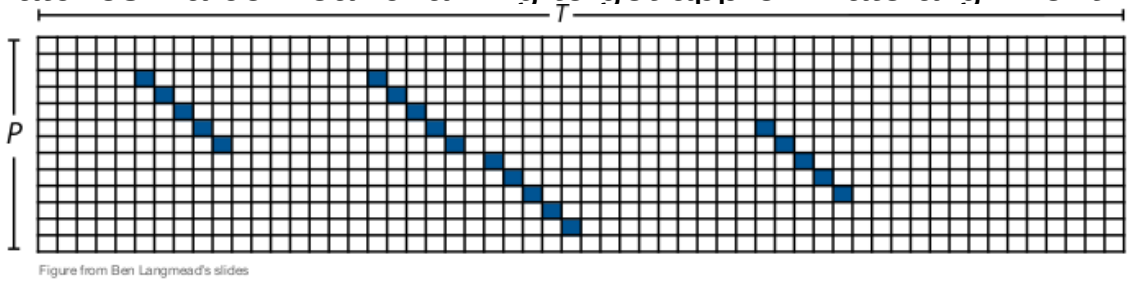
Presenter: Subhasree Majumder

Indian Institute of Science, Bangalore

19th May, 2021

Need for Minimizers

We had learnt in earlier classes regarding hueristic approximate matching wherein we find k-length exact matching substring called anchors also known as k-mers between two strings. And then we extend our the k-mer matches via colinear chaining to get approximate alignment.



Kmers (Ref: DS202 Lecture 18)

Given a set of N strings $\{T_i\}_{i=1 \text{ to } N}$ every k-mer in each string T_i is stored.

For example, string ATCGAT will have ATC, TCG,CGA,GAT as its 3-mers.

But are all the k-mers necessary for alignment of pair of strings? No. Only k-mers common in more than one string are needed, but how to find those common k-mers

The advantage we get by storing all k-mers in a database is that when we sort the kmers, we get identical k-mers adjacent to each other, thus giving us all the k-mers that we need to consider as ‘seeds’ or ‘anchor’ to apply the colinear chaining or ‘seed-and-extend’ method. This property of

But storing all k-mers is memory intensive

Length of string T_i = $|T_i|$

No. of k-mers = $|T_i|$ - k + 1 ~ $|T_i|$ since k << $|T_i|$

K-mer string s is stored as a triple tuple (s,i,p) where k-mer s appears in position p in string T_i.

Length of all strings |N| = L
So storage ~ O(k*L)

For exmple,

No. of reads = 33*10^6
|read| = 600
Hence, No. of k-mers ~ 2*10^10
|K-mer| = 20
Using compressed storage, we require around 10 bytes per tuple.
=> 2* 10^10 * 10 = 2*10^11 bytes ~ 200 GB

Minimizers

- In order to reduce k-mer storage, we can store **a subset of k-mers, called minimizers** but what should be the strategy of selecting that subset?
- **Strategy 1**: Store every k-th position k-mer so that each letter is covered once?

The problem here is that the two sequences containing common subsequence starting at pi and pj respectvely may not have a common k-mer stored if pi-pj is not a multiple of k.

A T G A A T C G T T A A {3-mers: ATG, AAT, CGT, TAA}
i=3

G T T C G A A T C {3-mers: GTT, CGA, ATC}
j=5

Strategy 2: Minimizers (also known as interior minimizers)

We choose a subset of k-mers such that they satisfy the following property:

Property 1: If two strings have a significant match then atleast one of the minimizers chosen from one will also be chosen from the other.

We look at a window of w consequtive k-mers and select the lexigographically (ordering) smallest k-mer as the minimizer. If more than one smallest k-mer, all of them are stored as minimizers. These minimizers are called (w,k)-minimizer.

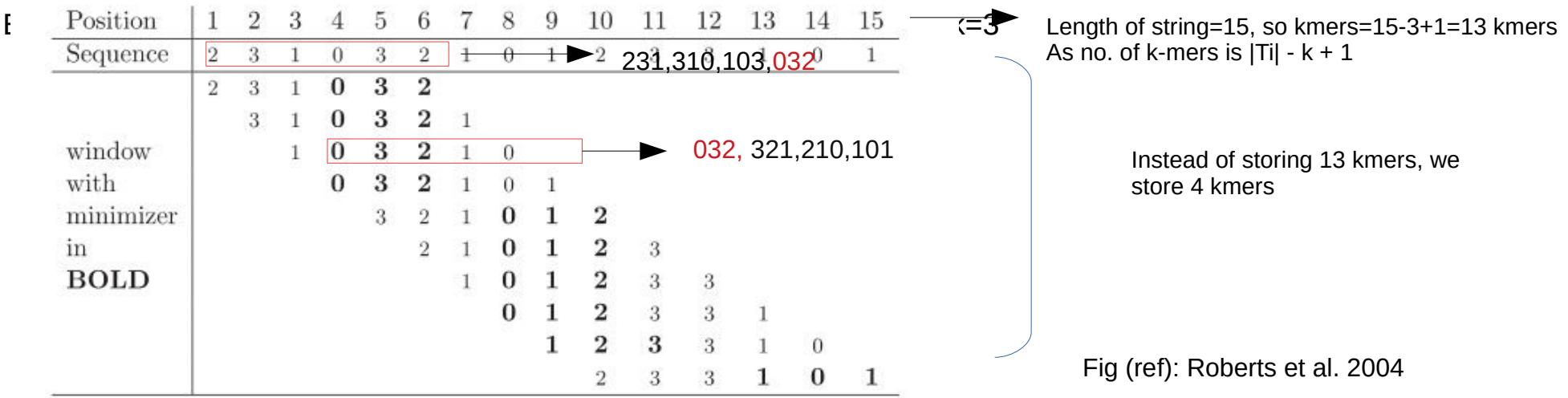


Fig (ref): Roberts et al. 2004

So we can restate Property 1 as: If two strings with w+k-1 matching characters will have a (w,k) minimizer in common

Gaps not covered by minimizers

We can see that minimizers did not cover all letters for example positions 1-3,7,12.

We note that minimizers in two adjacent windows can differ by atmost w positions because the first window's minimizer could be at the beginning kmer and the adjacent window's minimizer is at the wth i.e. the last kmer position. So gap size is atmost w-k.

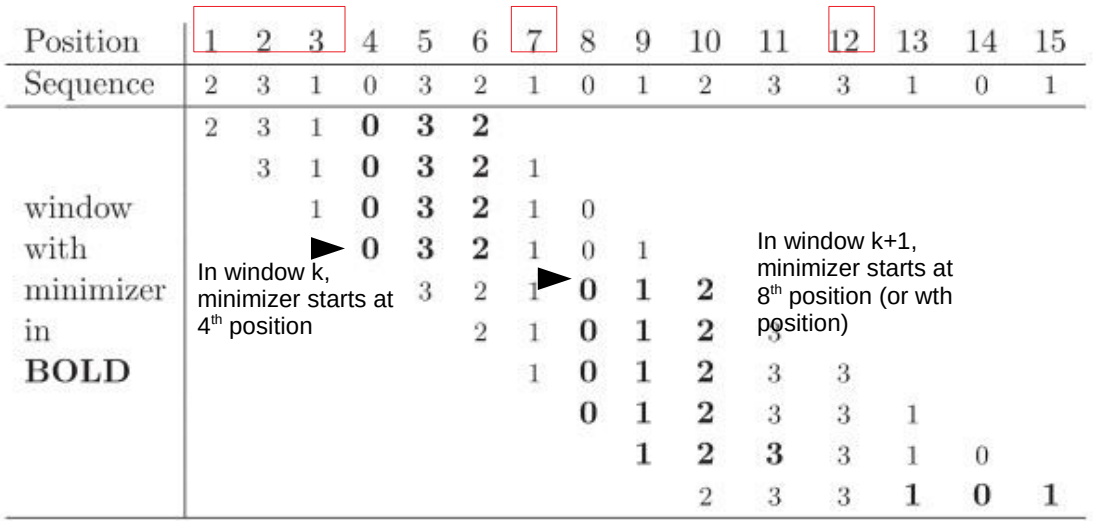
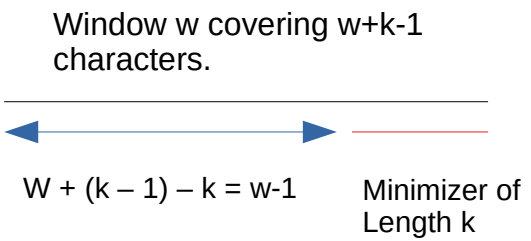


Fig (ref): Roberts et al. 2004

So setting w<=k ensures no gap occurs between minimizers except the **first and ending w-1 characters**. This happens when at the start of the sequence if the mnimizer of first window starts at the last kmer in w and consecutive windows do not have minimizers that cover these w-1 letters. Similarly this stuation might happen at the end of the sequence also.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sequence	2	3	1	0	3	2	1	0	1	2	3	3	1	0	1
	2	3	**1**	**0**	**3**										
		3	1	**0**	**3**	**2**									
			1	**0**	**3**	**2**	1								
				0	**3**	**2**	1	0							
					3	2	**1**	**0**	**1**						
						2	1	**0**	**1**	**2**					
							1	**0**	**1**	**2**	3				
								0	**1**	**2**	3	3			
									1	**2**	**3**	3	1		
										2	**3**	**3**	1	0	
											3	3	**1**	**0**	**1**



Strategy 3: End Minimizers

So we saw that even without gaps, minimizers might leave out from covering w-1 characters in the sequence ends. If two sequences match each other on their ends such that the length of the match is less than w+k-1. Then there is a possibility that there are no common (w,k) minimizer for the common portion of the string.

We define a (u,k)-end-minimizer where u is the size of window and u varies from 1 to some max size v. The end minimizers are anchored to the end.

But again end mi cters.

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Sequence	2	3	1	0	3	2	1	0	1	2	3	3	1	0	1	1
	2	**3**	**1**													
	2	**3**	**1**	0												
	2	3	**1**	**0**	**3**											
	2	3	1	**0**	**3**	**2**										
	2	3	1	**0**	**3**	**2**	1									
	2	3	1	**0**	**3**	**2**	1	0								
	2	3	1	**0**	**3**	**2**	1	0	1							
	2	3	1	0	3	2	1	**0**	**1**	**2**						
	2	3	1	0	3	2	1	**0**	**1**	**2**	3					
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3				
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3	1			
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3	1	0		
	2	3	1	0	3	2	1	**0**	**1**	**2**	3	3	1	0	1	

Here k-end minimizer wth k=3 and u=1,2,....l-k+1 where l is the length of string

Strategy 4: Mixed minimizers

We club together (w,k)-minimizers with (u,k)-end minimizers for u=1,...,w-1 at both ends of the string.

As w<=k, every character will be covered by minimizer.

The ends of the string will also be covered by end-minimizer.

Thus it guarantees that strings with matches at the end will also have common minimizers.

Other Applications of minimizers:

- **Sampling Suffix Arrays using minimizers**

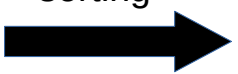
T = once_upon_a_time

Suffix_array:

Suffixes : T[j....|T|] , where $0 \leq j \leq |T|$

0 *once_upon_a_time*
1 nce_upon_a_time
2 ce_upon_a_time
3 e_upon_a_time
4 _upon_a_time
5 upon_a_time
6 pon_a_time
7 on_a_time
8 n_a_time
9 _a_time
10 a_time
11 _time
12 time
13 ime
14 me
15 e

sorting



9 _a_time
11 _time
4 _upon_a_time
10 a_time
2 ce_upon_a_time
15 e
3 e_upon_a_time
13 ime
14 me
8 n_a_time
1 nce_upon_a_time
7 on_a_time
0 once_upon_a_time
6 pon_a_time
12 time
5 upon_a_time

SA = {9,11,4,10,2,15,3,13,14,8,1,7,0,6,12,5}

The Algorithm:

T = once_upon_a _ t i m e

SA = {9,11,4,10,2,15,3,13,14,8,1,7,0,6,12,5}

1. Pass a sliding window of length w over T
2. Calculate the lexicographically smallest substring of length p in each window (the minimizer)
3. leftmost smallest string is chosen incase of ties
4. position of minimizers are the positions of the sampled suffix
5. lexicographically sort the sampled suffixes

Example:

w=5 ; p=3

T = once_upon_a_time

onc, nce, ce_ => ce_ => j=2

T = once_upon_a_time

nce, ce_, e_u => ce_ => j=2

T = once_upon_a_time

ce_, e_u,_up => _up => j=4

Similarly, Minimizers=> {ce_,ce_,_up,_up,_up,on_,n_a,_a_,_a_,_a_,_ti,_ti}

Suffix index => {2,2,4,4,4,7,8,9,9,9,11,11}

Sampled suffixes => {ce_upon_a_time, _upon_a_time, on_a_time, a_time }

T = once_upon_a _ t i m e

To match a pattern P in text T, ex: P = upon_a

1. We calculate minimizer in P[1...w] i.e. prefix of the pattern

   Say, w=5 => on_ is the minimizer found at position j=2
2. Binary search the suffix P[j...|P|] in the SA_new

3. Verify each match with the truncated (j-1) prefix of the pattern P.

The work by *Grabowski et al.* 2014 was able to search patterns of length 50 by 10% faster time with minimizer based SA over plain SA while the minimizer based SA only sampled 5.3% of the total suffixes. Thus a huge time and space optimization has been observed using minimizer based SA.

Other Applications of minimizers:

- **Fast and Low memory compaction of de-bruign graphs using minimizers**

Brief recap about de-bruign graphs and definitions:

1. de-bruign graphs represent the information from a set of reads. Given a set of reads R, every distinct k-mer in R forms a vertex of the graph and an edge connects two vertices if they have a k-1 suffix-prefix overlap.

2. Compaction of debruign graphs help in data reduction wherein long unitigs are compacted into single vertices.

3. A unitig is a path p =(x_1,...,x_m) over m≥vertices such that |p|=1 or 1<i<m and out and in degree of x_i is 1 and the in-degree of x_m is 1 and out degree of x_1 is 1.

4. We call two strings u and v compactable, if u is the only in-neighbour of v and v is the only out-neighbour of u. The compaction operation replaces u and v with $u.^{k-1}v$.

5. However compaction is a memory intensive step. As we need to load the entire graph into memory, compact them iteratively until no further compaction is possible.

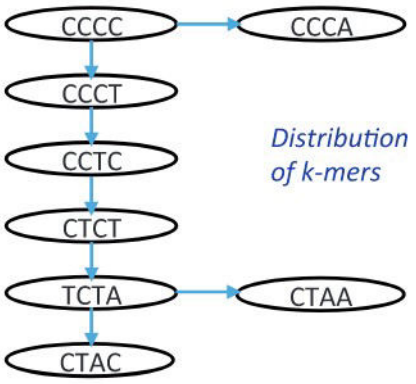


Fig: De-bruign graph (Ref: R Chikhi et al. 2016)

The Algorithm (BCALM2):

Pseudo Code 1:

Input: set of k-mers K from reads.
1. for all x ∈ K do
2. write x to F(lmm(x))
3. if lmm(x) ≠ rmm(x) then
4. write x to F(rmm(x))
5. for all parallel i ∈ {1,...,4^l}
6. Run c

lmm(x) is the l-minimizer of k-1 prefix of x

rmm(x) is the l-minimizer of k-1 suffix of x

l is the size of minimizer. k is the size of k-mer length vertices in debruign graph

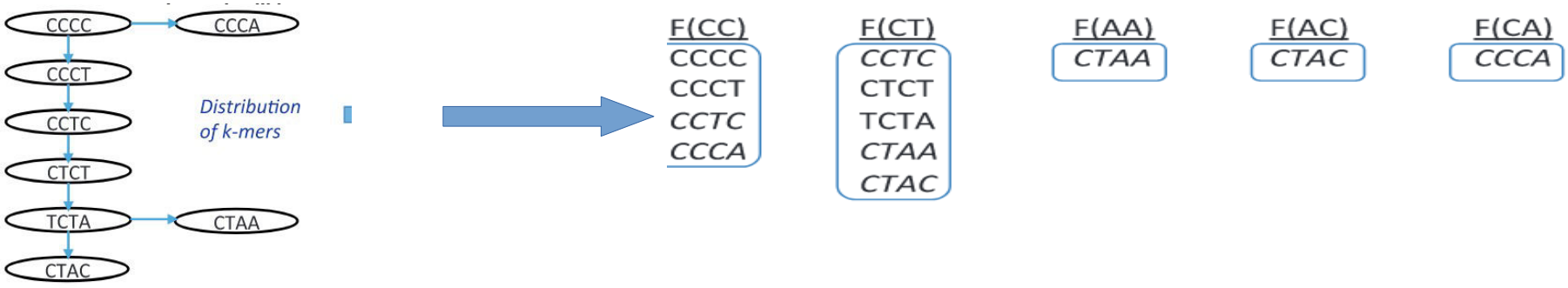
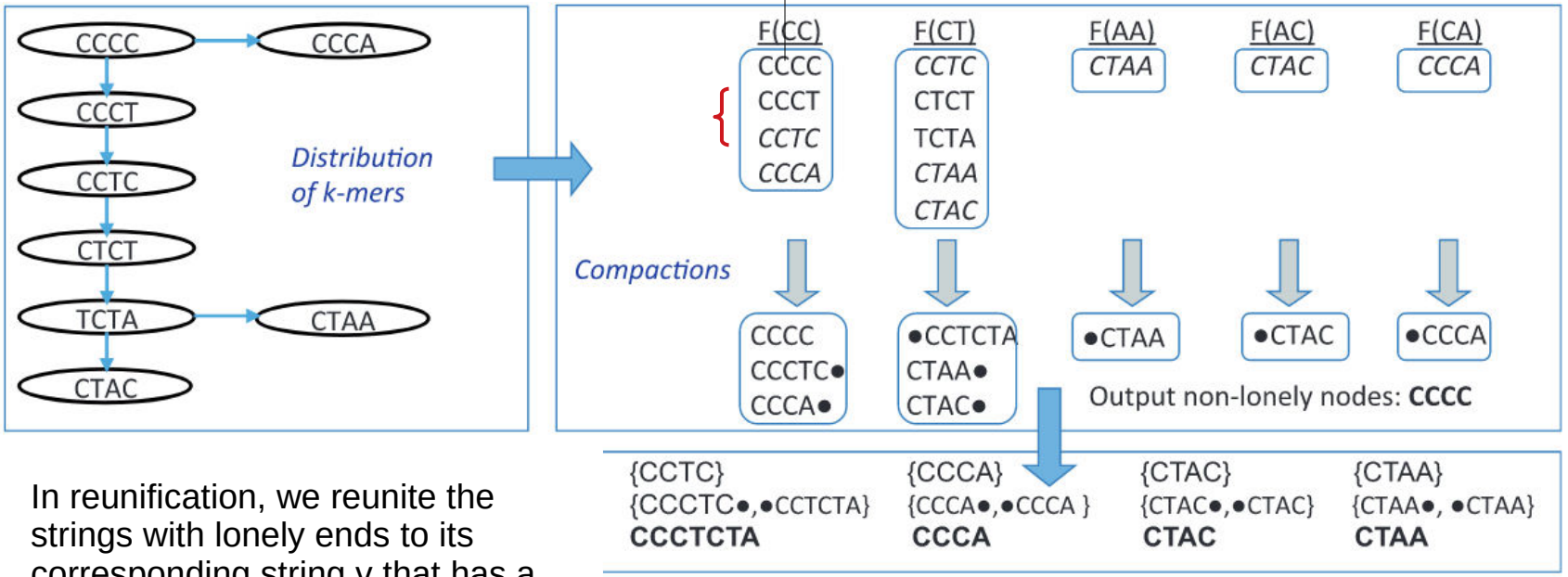


Fig: Distribution of k-mers to F(lmm) buckets (Ref: R Chikhi et al. 2016)

Pseudo Code 2:

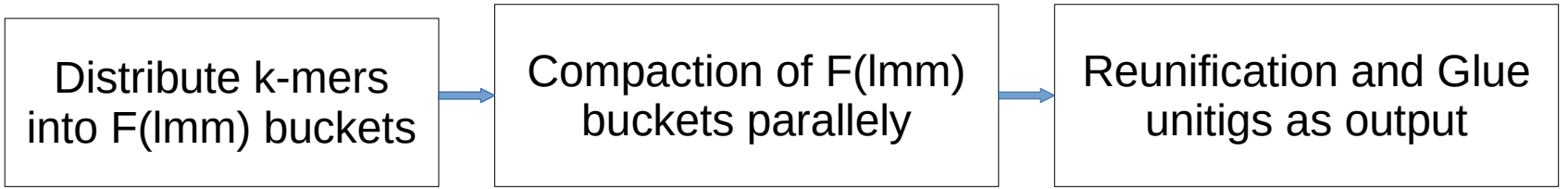
1. Load F(i) into memory
2. U <- compact u and v in F(i) to $u.^{k-1}v$
3. for all string u ∈ U do
4. Mark u's prefix as lonely (.) if i ≠ lmm(u)
5. Mark u's suffix as lonely (.) if i ≠ rmm(u)
6. if u's prefix and suffix are not lonely then
7. output u
8. else
9.

CCCC is not compacted as it has two out-neighbours CCCT and CCCA violating compactable condition



In reunification, we reunite the strings with lonely ends to its corresponding string v that has a matching lonely end. We glue them to their outputs.

BCALM2: Fast and Low memory compaction of de-bruign graphs using minimizers



Performance: The authors [R Chikhi et al.] noted BCALM2 compact debruign graph construction took 76 mins and 3 GB memory on human genome sequence surpassing SOA tools Abyss-P (6.5h,89GB)

Experiments : Effect of window size w and k-mer length l on accuracy of read mapping using minimizer based tool minimap2

Type of Analysis	w	k	Number of read hits	mapQ=0	mapQ≤50	mapQ>50	Runtime (sec)
Constant w Increase k w<k	3	10	127	10	1	116	1.69
	3	15	123	6	0	117	0.38
Increase w Constant k w<k	10	20	117	0	0	117	0.24
	15	20	117	0	0	117	0.25
w=k	20	20	117	0	1	116	0.26
	50	50	114	0	0	114	0.26
Constant w Increase k w>k	100	15	115	2	3	110	0.33
	100	25	113	5	6	102	0.43
Increase w Constant k w>k	200	28	94	12	8	74	0.58
	255	28	80	7	12	61	0.56

$mapQ = 40.(1-f_2/f_1) \cdot Min\{1,m/10\}.\log f_1$ [Li Heng minimap2]

Reference Dataset: Ecoli
Length: 4 million bp
Average read length~3000bp
Reads Instrument: MinION
Reference:
http://s3.climb.ac.uk/nanopolish_tutorial/ecoli_2kb_region.tar.gz

Mappy: https://pypi.org/project/mappy/

References:

[1] Roberts, Michael, et al. "Reducing storage requirements for biological sequence comparison." Bioinformatics 20.18 (2004): 3363-3369.

[2] Grabowski, Szymon, and Marcin Raniszewski. "Sampling the suffix array with minimizers." International Symposium on String Processing and Information Retrieval. Springer, Cham, 2015.

[3] Chikhi, Rayan, Antoine Limasset, and Paul Medvedev. "Compacting de Bruijn graphs from sequencing data quickly and in low memory." Bioinformatics 32.12 (2016): i201-i208.

[4] Li, Heng. "Minimap2: pairwise alignment for nucleotide sequences." Bioinformatics 34.18 (2018): 3094-3100.

[5] “Minimizers – an introductory tutorial”, 25th Oct, 2017, accessed 10th May, 2021
https://homolog.us/blogs/bioinfo/2017/10/25/intro-minimizer/

Questions, Feedback and Comments

Thank you :)