

Program File Tree Structure and Auxiliary Files

Introduction

```
|— src # source code
|   |— res_store # the dir contains all the result for application
|   |   |— corrected_reads.txt
|   |   |— true_reads.txt
|   |   |— mutated_reads.txt
|   |   └─ ... and many other results
|   |— contamination.py
|   |— correction.py
|   |— application.py
|   |— local_alignment.py # to do the alignment
|   └─ plot.py # to plot
|— test_cases
|   |— application
|   |— correction
|   └─ contamination
|— images
|   |— img_for_application
|   └─ img_for_readme
|— README
|— correction.sh
└─ contamination.sh
```

Shell Command

When run any command, make sure you are under the corresponding directory.

Specifically, here, you need to under the root dir.

Contamination

```
sh contamination.sh <reads.txt> <vector.txt> <k>
```

```
-----
sh contamination.sh ./test_cases/contamination/contam_reads1.txt
./test_cases/contamination/vector1.txt 3
```

Correction

In order to pass the autograder command line, the command line does not add a parameter to choose the algorithm.

However, notice the writer implemented multiple ways to apply the correction and they will be discussed in the Implementation part.

```
sh correction.sh <reads.txt> <k> <t> <d>
```

```
-----  
exm: sh correction.sh ./test_cases/correction/error_reads1.txt 2 2 1
```

Implementation

Contamination

Implemented the contamination part as instruction.

Correction

Overall

The writer implemented four methods to deal with correction and are listed as following:

1. Stack correction (default mode)
2. Simple correction
3. Navie correction
4. Merge correction (please check this one, I want to discuss it if you are interested!)

No matter which method, the key idea is solve the overlapping problem and try to locate the index with the highest possibility that has error.

Since the errors in the sequence is only 1%, and they are approximately uniformly distributed. Therefore, it is not likely we have multiple errors in a continous range.

Stack Correction

As the name implies, the writer uses stack data structure to implement this method.

The high-level algorithm works as the following:

- Iterating the error reads for i times, for each read:
 - Initialize a stack (remember stack is First-in-Last-out data structure), the stack will store the start index of the infrequent k-mer we need to correct
 - Iterating the index in the current read
 - if the current k-mer is infrequent, peak the stack:
 - if last element in stack gives us an index so that the distance between it and current index is smaller than k , we ignore current infrequent k-mer since they are too close.
 - otherwise, we push current index into stack and keep going.
 - after we finishing iterating current read and the stack, we go through the element in the stack and find the target index we need to correct and correct the read according to the instruction method
 - after updating all of the reads in the error reads, update the frequent kmers, infrequent kmers and other related structures.

Now let's have an example:

$k = 4$

read index =	0	1	2	3	4	5	6	7	8	9
freq notation (0: freq):	1	0	1	1	0	0	1			

	stack	current idx	check
init	$[-\infty]$	init	/
1st 1	$[-\infty, 0]$	0	$0 - (-\infty) \Rightarrow Y$
2nd 1	$[-\infty, 0]$	2	$2 - 0 < 4 \Rightarrow N$
3rd 1	$[-\infty, 0]$	3	$3 - 0 < 4 \Rightarrow N$
4th 1	$[-\infty, 0, 6]$	6	$6 - 0 > 4 \Rightarrow Y$

∴ finally, we will have:

read index =	0	1	2	3	4	5	6	7	8	9
freq notation (1: freq):	1	0	0	0	0	0	1			

and we need to correct the infrequent k-mer on index 0 and index 6.

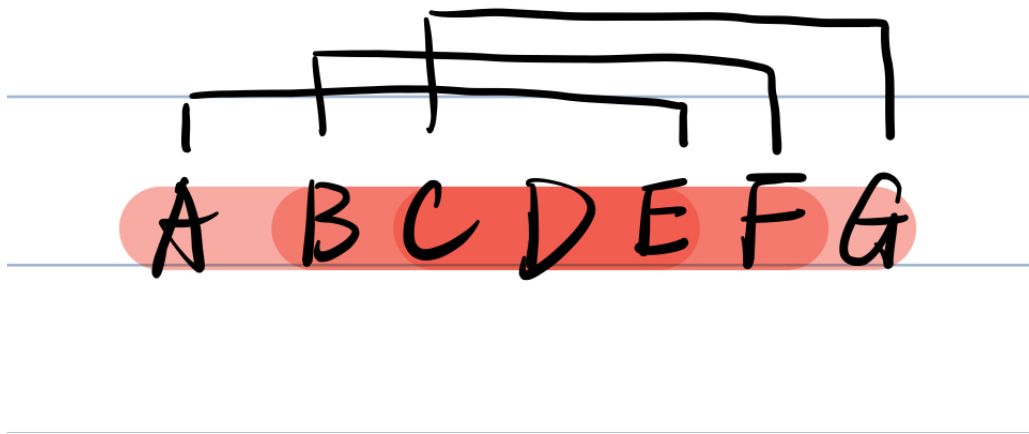
Simple Correction

Simply correct all the infrequent k-mer we get, no matter how close they are with each other, just correct them from left to right.

This naive correction is designed to have a comparison between other methods, since the writer is curious about how exactly this kind of "uniformly distributed error" affects the system. And is there possibility by selecting the potential correct frequent kmer, even if they are overlapped, the error can be corrected and the correct base will not be affected.

Naive Correction

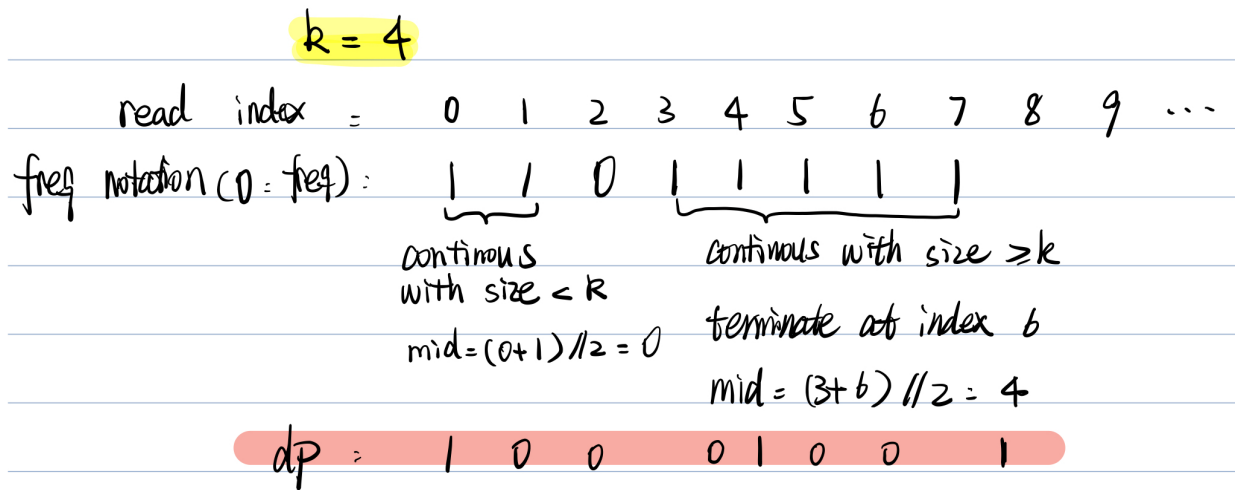
Intuitively, what this algorithm is doing is when there are continuous infrequent k-mers, choose the middle one, cause when you considering overlapped, continuous k-mers, as you can see from the following figure. The bases in the middle will have the highest possibility to have the error. And by choosing the middle k-mer, we can cover the bases with highest possibility.



The algorithm is implemented by dynamic programming, it is easy to understand and think of, however, there are two situations we need to pay attention:

1. number of continuous infrequent k-mer smaller than k
 - for this case, we will simply choose the middle one as infrequent, and denote other position to frequent
2. number of continuous infrequent k-mer larger than k
 - for this case, we will cutoff the continuous infrequent k-mer with size k, and mark the middle one as infrequent and others as frequent.
3. Then we iterating the dp and correct the position with 1.

Let's see an example:



Merge Correction

I personally LIKE this algorithm, however, I do not have time to test it on the hiv genome, but I will keep update the results in the github repo if I finish running it.


Intuition

Lets start with some examples, there are basically **three situations**:

 : infrequent  : frequent

① two infrequent k-mer is not overlapped.

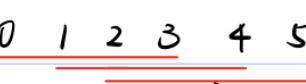
read index : 0 1 2 3 4 5 6 7



⇒ Our Best try : correct infrequent ones sequentially

② two infrequent k-mer is continuous with no freq. gaps.

read index : 0 1 2 3 4 5 6 7

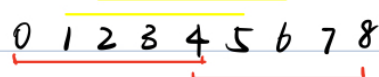


⇒ Our intuitively Best Guess ?

- There might be something wrong with index 2 & 3
 - overlapped by 3 times (more than other idx)
 - errors are rare : narrow down to only 2 index

③ two infrequent k-mers with frequent gaps in between.

read idx : 0 1 2 3 4 5 6 7 8



⇒ Our intuitively Best Guess ?

Something wrong with index 0 & 8

since index in between gives us frequent k-mers

Algorithm

Now, if we want to follow this kind intuition, how to make our program detect the target index?

we can take a vote and restore the vote with dynamic programming!

- in the range of frequent k-mers, each base get "vote" -1
- in the range of the infrequent k-mers, each base get vote +1
- Finally, we are looking for the index(s) have **highest votes** or higher than some threshold (I take the first choice to min error number)

Ok, now let's try it on the above real example, since for the first case, we correct directly, let's see the latter two:

②	
read index :	0 1 2 3 4 5 6 7
vote dp :	1 2 3 3 2 1 0 0
③	
read idx :	0 1 2 3 4 5 6 7 8
vote dp :	1 0 -1 -2 -1 -2 -1 0 1

It seems we get what we need! Now what...?

- If we get continuous target index like two 3 in situation 2, we want to replace them together.
- if the index are separate, just deal with them separately.

Objective

Now suppose we have a target result need to correct, we denote as $[i, j]$, where i is the start index, and j is the end index, and suppose this is an h -mer with length h . Then we want to find the frequent h -mer that can maximize the number of frequent k -mers starts from index $i - k + 1$, ends at index j .

- **build a cache:** Notice the re-building h -mer may be time consuming especially when we cannot ensure what length of h -mer we can get, therefore, the writer create a hash map structure to store the built frequent and infrequent h -mer with key as length, value as mapping, such that, when encountering the same length h -mer again, we can directly use the result.
- **ignore distance:** this algorithm actually ignore the distance threshold we have in the instruction, why? consider the case in error read 1, there is a read "AAAAAGGAA", when apply this algorithm, the position we need to correct will become "GG". This is intuitive, since after the voting, we are breaking the framework of k -mer, but only focusing on the highest possibility locations. Now the distance threshold for k -mer is meaningless if you consider. Our optimization objective is changed to what we mentioned above.
- **how to improve enumerating:** as we know, the vanilla idea of the optimization objective can be enumerating the possible h -mers and get the best one. This is really time consuming, how to improve it? Two ideas are used:

- o **more frequent, more correct:** as the idea in instruction, the frequent $h - mer$ has higher possibility to be the correct answer, therefore, we want to start from the most frequent $h - mer$ and gradually decrease the frequency. What data structure we can use? Yes, heap! A perfect data structure for dynamically calculate the k-largest with $O(n \log n)$.
- o **early stop:** I also borrow the idea from the deep learning: when training the model, if the performance is not increase in certain epochs, stop the training process. So in our case, it would be, starting from the most frequent option to least frequent option, if the number of frequent $k - mer$ is not increasing for certain tryouts, then stop and take the current optimal one.

Problem

Ok, so what is the problem of this algorithm:

- large time complexity: the "enumerating" and find the target giving largest number of frequent k-mers is time consuming
- intuition might not be right
- large space complexity: storing cache we need extra large space especially when h diverse largely.

Test Sample

Contamination

Test case passed

```
Input: sh contamination.sh contam reads1.txt vector1.txt 3
```

Got :

0, 1, 2, 3

TGCCCTG
CCCTG
TGCC
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Figure S7. The DNA sequence of the CRISPR array.

Expected:

0, 1, 2, 3

[illegible]

Test case passed

Input: sh contamination.sh contam_reads1.txt vector1.txt 4

Got:

0,1,2,3

```
AAATGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCTGAAA
AAATGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Expected:

3

```
AAATGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCTGAAA
AAATGCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Correction

Correction Test 1

Test case passed

Input: sh correction.sh error_reads1.txt 2 2 1

Original Reads:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAGGAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAATTAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAATAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACAAAAAAA
```

Infrequent indices:

```
AAAAAAAAAAAAACCAAAAAAAAAAATTAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

4 5 6

Got:

4

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAATTAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAATAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACAAAAAAA
AAAAAAAAAAAAACCAAAAAAAAAAATTAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Correction Test 2

Test case passed

Input: sh correction.sh error_reads1.txt 5 3 2

Original Reads:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACCAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAGGAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAATTAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAATAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACAAAAAAAAA
```

Infrequent indices:

```
AAAAAAAAAACCAAAAAAAAAATTAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

36 37 38 39

1 2 3 4 5 6

11 12 13 14

25 26 27

38 39 40

8 9 10 11 20 21 22 23

Got:

1,4,5,6,8,9

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Bug Reporting

Currently no identifying bugs