Stockholms universitet

Improving indexing for computational error correction of RNA sequences with syncmers

Sebastian Lindbom Gunnari

Svensk titel: Förbättring av indexering för felkorrigering av RNA-sekvenser med syncmers

Handledare: Alexander Petri and Kristoffer Sahlin
Examinator: Lars Arvestad
Inlämningsdatum: 2022-01-06

Kandidatuppsats i datalogi
Bachelor Thesis in Computer Science

Abstract

In computational genetics it is common practice to store genetic sequences as substrings of length k called k-mers. To increase efficiency of these k-mers, indexing techniques that allow for more critical selection are used. One such technique is minimizers, a subsampling technique that is commonly used in genomic tools for reducing memory and improving speed. However, in 2021 a new indexing technique called syncmers was suggested by Robert Edgar, with promises of increased performance over minimizers. Here we show that syncmers have the potential to increase performance for the error correction tool isONcorrect, which currently uses minimizers. Under the conditions we ran our tests, syncmers were shown to be more resistant to k-mers being destroyed when errors were introduced into a sequence. They were shown to cover a larger proportion of sequences due to being more spread out across the sequence string. An interval scoring system used by isONcorrect also shows that syncmers are better than minimizers at conserving intervals of k-mers. Although the analysis done in this paper is not exhaustive, we anticipate that it will motivate further research into the advantages and pitfalls of syncmers so that their full potential can be utilized in the future.

Sammanfattning

Inom beräkningsgenomiken är det praxis att lagra gensekvenser som delsträngar av längd k. För att förbättra effektiviteten hos dessa används indexeringstekniker som tillåter ett mer kritiskt urval delsträngar. En av dessa tekniker är minimizers, en delsamplingsteknik som ofta används i genomiska verktyg för att minska lagring och förbättra hastighet. 2021 föreslog dock Robert Edgar en ny indexeringsteknik som han kallar syncmers, vilken han visar kan överträffa minimizers i vissa avseenden. Vi visar i denna uppsats att syncmers har potential att förbättra prestandan för isONcorrect, ett verktyg för felkorrigering av gensekvenser som för närvarande använder minimizers. Givet de förhållanden vi utförde analysen under visade sig syncmers vara mer resistenta än minimizers mot att förstöras när felaktigheter introduceras i sekvenser. De visade sig också täcka större andelar av sekvenserna och vara mer utspridda över sekvenssträngarna. En av isONcorrects interna algoritmer visar också att syncmers är bättre än minimizers på att bevara viktiga intervall av delsträngar. Analysen i denna uppsats är inte uttömmande, men vi förväntar oss att den kan motivera fortsatt forskning kring syncmers för- och nackdelar så att deras fulla potential kan utnyttjas i framtiden.

Contents

1 Introduction

The last decades have seen an exponential increase in genetic information available to researchers. Various indexing techniques have been used to improve the problem of storage. One such technique called Minimizers, was first suggested in 2003-2004 as a subsampling technique to reduce space [3, 5]. Methods such as the 'seed-and-extend' approach of string matching worked with sequence databases so large that the 'seeds' (substrings) could not be stored in the memory of a single computer. Compared to a more inefficient approach of storing all substrings of a sequence, minimizers allow for a predictable but more scrutinizing selection of substrings with only minor losses in effectiveness for the applications they are used.

Minimizers have been central to such widely used applications as minimap2, which allows searching for matches of genetic sequences by aligning and comparing them to a large database [2]. Recently it has been utilized in isONcorrect, which is a tool for correcting Oxford Nanopore cDNA reads [4]. Oxford Nanopore Technologies (often shortened to ONTs) has been revolutionizing the field by allowing for long end-to-end reads, albeit with high error rates. Tools such as isONcorrect will certainly play an important part in allowing ONTs and similar long read technologies to reach their full potential.

In 2021, Robert Edgar wrote a paper suggesting a new type of indexing technique, which he calls syncmers [1]. In his paper, he shows that syncmers are not as sensitive to mutations and error reads as minimizers are. In this thesis, we will compare the performance of syncmers to that of minimizers in a few different aspects that are important to the effectiveness of isONcorrect.

2 Methods

2.1 Selection based on k-mers

The basic principle behind the different local selection methods relies on dividing strings into substrings of length k commonly known as "k-mers", and selecting or rejecting these k-mers based on some pre-defined criteria. This allows for a repeatable and predictable selection of substrings. More formally, we can define selections based on k-mers to be functions over a finite set of substrings to a set of tuples (x, i), where x is a k-mer and i represents the start position of the k-mer in the string [6].

For the selection criteria, a total order is used. A common total order used for sequence based analysis is the lexicographical order. This means hashing the letters of the k-mers into numerical values and performing a selection based on their order. Ties may be resolved by picking the leftmost of the tied k-mers [6]. We are using the lexicographical order exclusively in this thesis. References to order are from now on assumed to be lexicographical.

In this thesis, two different indexing techniques called minimizers and syncmers are compared. They use the common terminology of k-mers, and are both based on selecting k-mers based on the chosen order of substrings. In the case of minimizers, the k-mers within a window are compared to each other according to the chosen total order, and the minimal k-mer with respect to this order is selected. Syncmers instead make use of s-mers, with $s < k$, which are substrings of the k-mers themselves. These s-mers are evaluated both by their position and their chosen order within an individual k-mer. If an s-mer holds up to the selection criteria, the k-mer which encloses it is selected. This will be explained further below.

2.2 Minimizers

The minimizer indexing technique is based on selecting k-mers from groups of consecutive k-mers defined by so called windows. A window defines a group of w consecutive k-mers, from which one is selected based on the chosen ordering. This process then continues with the sliding window moving across the string one base pair at a time and one k-mer being selected from each window, until it reaches the end of the string.

The minimizers can be characterized by tuples (w, k), giving the window size w and the k-mer size k. They are also denoted as tuples $(w, k, \mathcal{O})$, with $\mathcal{O}$ being the chosen order used to select a k-mer from a window [6]. Since the chosen order is already defined to be lexicographical only (w, k) will be given here.

To generate minimizers from a given sequence S, if $S(a, b) \subseteq S$ is the substring starting at index a and ending at index b, the minimal substring of length k is selected from each string $S(m, w + k - 1 + m)$ for $m = 0, 1, 2, \ldots$ until the end of the string is reached. An example of this selection can be seen in Figure 2.1.

2.3 Syncmers

The syncmer method is another k-mer selection method making use of substrings of k-mers called s-mers [1]. Like minimizers, syncmers can be characterized by tuples. The tuple (k, s) will here define the parameters k for k-mer length and s for s-mer length. The s-mer is a substring of a k-mer, so $s < k$. Open syncmers also include the positional parameter t, which will be discussed further down in the text.

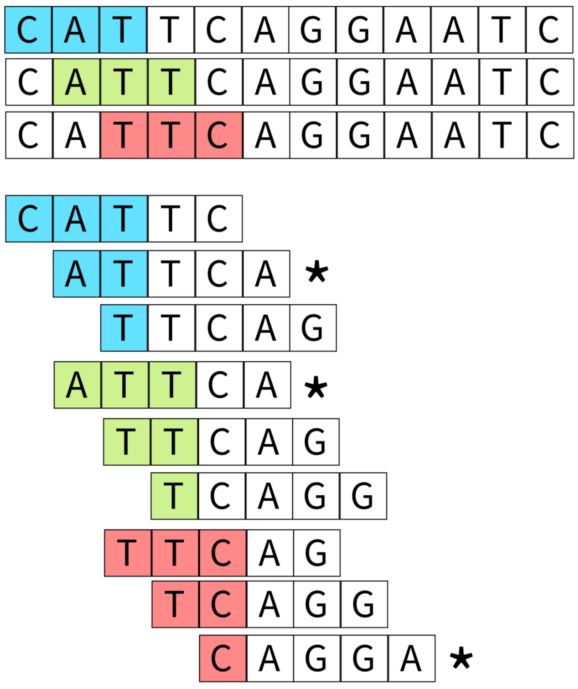

Figure 2.1: **An example for minimizers using (3,5).** This figure shows the first three minimizers using $(w, k) = (3, 5)$. The colors indicate separate windows, and the asterisks indicate the lexicographically smallest k-mer within each window. In this random sequence the k-mer ("ATTCA", 2) is selected twice, so the only k-mers stored in this selection would be ("ATTCA", 2) and ("CAGGA",5). Note that the sliding window would normally continue until the end of he string which is not shown here.

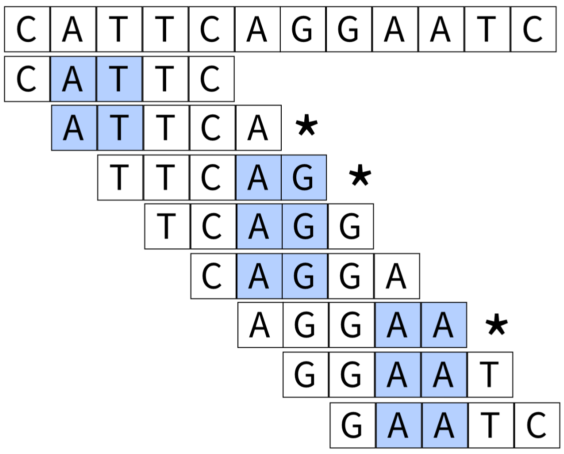

Figure 2.2: **Example for closed syncmers using (5,2):**. The asterisks mark the k-mers in which the smallest s-mer is at the beginning or end of the string. These k-mers are selected as closed syncmers.

The relationship between k-mers and s-mers is somewhat similar to the one between windows and k-mers in the minimizer method. The k-mers themselves can be seen as sliding windows moving across the sequence string, and the s-mers within the k-mers are selected according to the chosen order. The difference is, in the case of syncmers the s-mer order and position within the k-mer decides whether the k-mer is selected or not. If the s-mer fulfills the selection criteria within a k-mer, the k-mer it is contained in is selected.

While this thesis only focuses on comparing open syncmers to minimizers, we also describe the concept of closed syncmers.

Using a closed syncmer selection defined by (k, s), a k-mer is selected only if either the first or last s-mer hashes to the smallest value of all s-mers within the k-mer. An example run on an arbitrary sequence of base pairs is demonstrated in Figure 2.2.

Open syncmers also use a parameter t, which indicates the position the

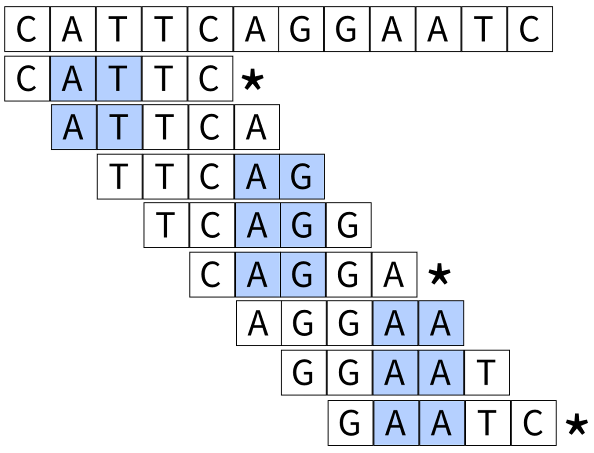

Figure 2.3: **An example for open syncmers using (5,2,2).** The asterisks mark the k-mers in which the smallest s-mer is at position t = 2 (using 1-indexing). These k-mers are selected as open syncmers.

minimal s-mer should be located at within a k-mer for it to be selected. Similarly to closed syncmers, the tuple (k, s, t) describes k-mers of size k which are chosen only if the minimal s-mer within the k-mer is at position t. This is illustrated in Figure 2.3 using the same sequence as the closed syncmers in Figure 2.2. Note that the sets of selected k-mers are disjoint.

Unlike minimizers, syncmers have no guarantee to select an open syncmer in any local area of a string. Consider for example a string $AAAAAA\ldots$ with $t = 2$. Since ties are resolved by choosing the leftmost k-mer, the "smallest" s-mer in any k-mer will be at position $t = 1$ and no syncmer will be found in this string.

2.4 Context-free versus context-dependent

A notable distinction between minimizers and syncmers is that the selection of a k-mer using minimizers depends on every k-mer within its window, while for syncmers it only depends on the k-mer being evaluated. If errors or mutations arise within the scope of a window in minimizer selection, the order of the k-mers within that window may be completely changed. On the other hand, syncmers should not be as sensitive, as only changes within the k-mer affects the selection criteria. Minimizers are therefore described to be context-dependent as opposed to syncmers that are described as context-free [1].

2.5 Implementation

This section describes the structure of the syncmer generation algorithm. A pseudocode showing the algorithm used can be seen below. It is based on using a queue filled with all the s-mers of the current candidate k-mer, which then moves in one base pair increments over the sequence. Moving from one candidate k-mer to the next is then simply a matter of removing the s-mer in front of the queue and adding the next one to the back.

INPUT: String S, Integer k, Integer s, Integer t
OUTPUT: A list of tuples (String k-mer, Integer index) describing the content of each k-mer together with the index at which it was found in the original sequence string

```
GET SYNCMERS($S, k, s, t$):
1. $w \leftarrow k - s + 1$
2. syncmers ← empty list
3. s-mers ← queue(S[$i : i + s$] $\boldsymbol{for}$ $i \in$ [0:w-1])
4. $\boldsymbol{for}$ $i \in [0, |S| - k]$:
5.        $\boldsymbol{if}$ argmin$_i$(s-mers) == $t$:
6.                tuple ← (S[i:i+k], argmin$_i$(s-mers))
7.                syncmers.add(tuple)
8.        remove the first element of s-mers
9.        enqueue S[$i + w : i + w + s$] to s-mers
10. $\boldsymbol{return}$ syncmers
```

As can be seen from the pseduocode, the run time for this algorithm is in $\mathcal{O}(n)$.

2.6 Setup of test conditions

The sequence strings used in this analysis are generated by a simple script. This script generates a string of given length, consisting of the characters 'A','C','G','T' selected (pseudo-) randomly at each position. To simulate the occurrence of erroneous reads and mutations, another script is used to introduce random errors into these sequences. Errors are simulated at an error rate of 1-10% in 1% increments. Each test is is repeated in 10 different iterations for each error rate. This means that a new sequence and its syncmers and minimizers are generated 10 times for each step between 1% and 10%. While this setup does not recreate a realistic scenario of real life sampling and errors, it is a simple approximation that suffices for the purposes of this analysis.

For a balanced comparison between the two indexing techniques, we decided that parameters s and t for syncmers should be found so that the number of resulting syncmers is close to the number of resulting minimizers. This was done experimentally by generating a large number of sequences and using the average number of k-mers yielded by either method to calculate the ratio. The inputs, parameters and ratio of syncmers to minimizers is shown in Table 2.1.

Number of base pairs in each sequence	10000
k	15
w	10
s	11
t	1
Number of iterations averaged	10
Average number of minimizer k-mers	1897.13
Average number of syncmer k-mers	1860.06
Syncmers/minimizers ratio	0.98

Table 2.1: **Inputs, parameters and number of syncmers and minimizers generated** Setting (k,s,t) to be (15,11,1) and (15,11,4) both gave a ratio with $|0.02|$ difference to the one containing minimizers (w,k) = (10,15). Therefore, (11,1) was arbitrarily chosen for the rest of the analysis.

3 Results

3.1 Conservation of k-mers

We say that a k-mer is conserved if we find the same k-mer in both the original and erroneous sequences. Measuring the ratio of conserved k-mers over original k-mers gives an idea of how sensitive either method is to losing k-mers as errors are introduced.

To find the proportion of conserved k-mers, minimizer and syncmer k-mers are generated for both a sequence and its error filled counterpart. The number of k-mers that are shared between the original and the error ridden sequence is then counted for either method. The proportion is found by dividing the number of shared k-mers by the average number of k-mers between the original and erroneous sequence. The box plots in Figure 3.1 show the spread of the proportion of k-mers conserved.

The results of Figure 3.1 seem to slightly favor syncmers, with the bulk of preserved syncmer k-mers scoring slightly higher than minimizer k-mers at most error rates. However as can be seen at error rates 1, 4 and 6 percent the difference between the two methods appear to be slight, perhaps with a bit of a larger spread for minimizers. At 1, 7 and 10 percent there are even outliers that outscored their syncmer counterparts.

3.2 Distribution of k-mers

Another important aspect is how well the syncmers/minimizers are distributed over the sequence, and which portion of the syncmers/minimizers is conserved in the erroneous sequence. To calculate this, the number of

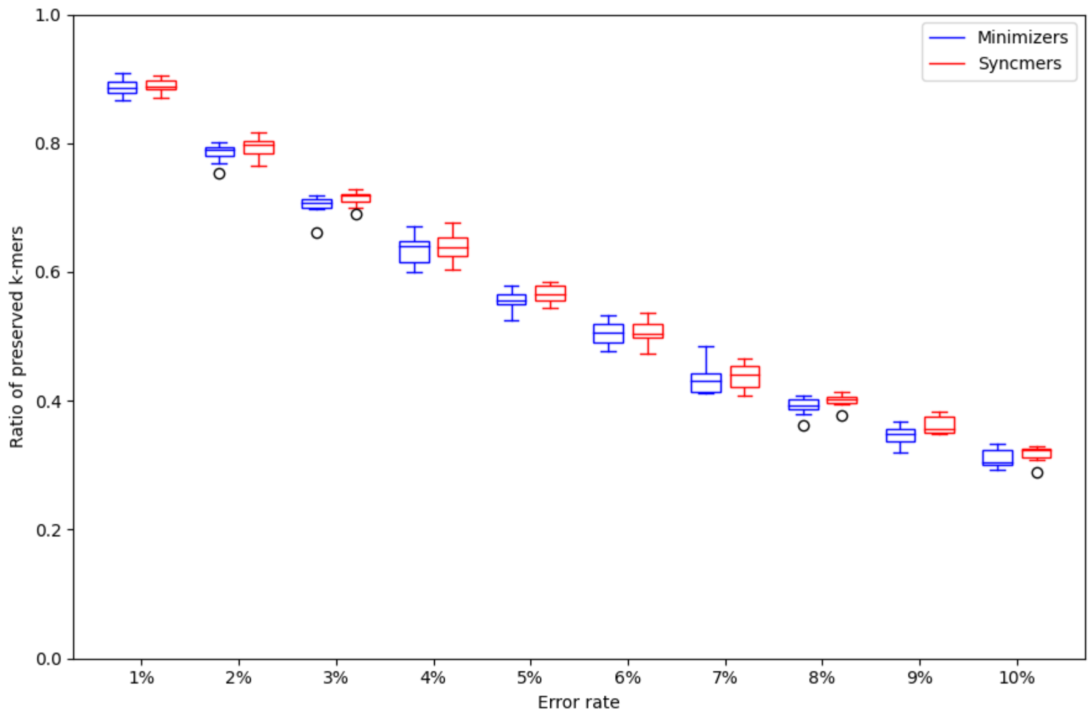

Figure 3.1: **Proportion of conserved k-mers.** A k-mer is considered conserved if it is selected both in a sequence and its error riddled counterpart.

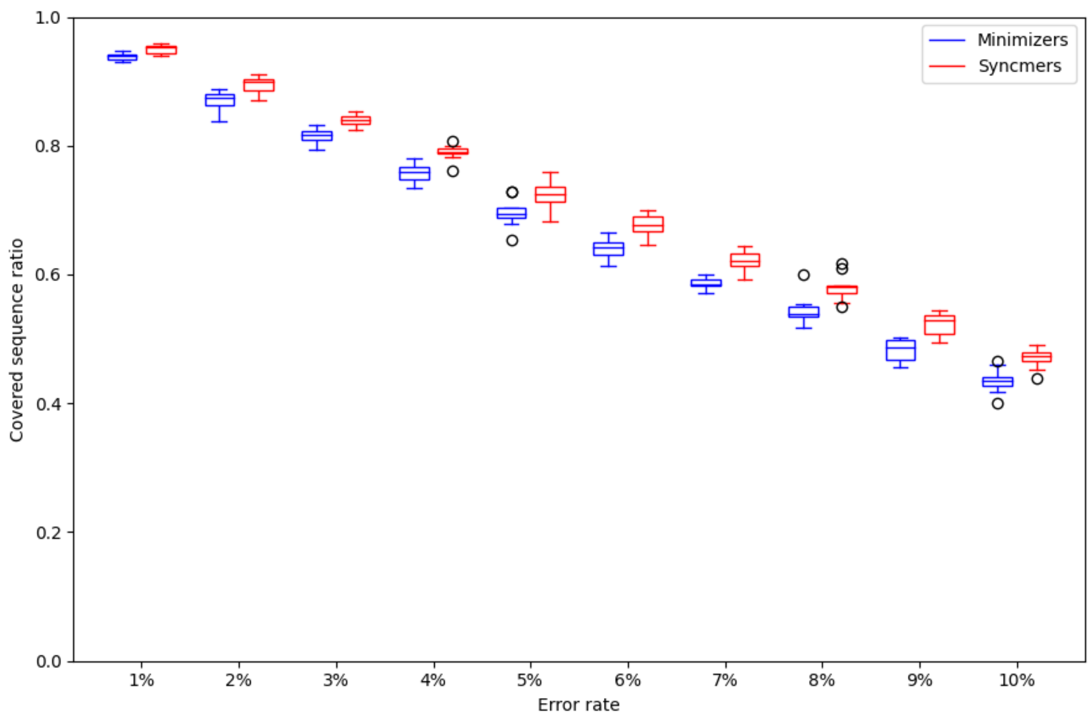

Figure 3.2: **Proportion of sequence covered by conserved k-mers.** That is, the ratio of the number of base pairs in the original sequence over the number of base pairs covered by conserved k-mers.

base pairs covered by conserved k-mers excluding overlaps is divided by the length of the original sequence. The results are illustrated in Figure 3.2, which is visibly skewed in syncmers favor. In some places such as at 8% error rate, outliers of minimizers performed very well.

We also asked how many base pairs each syncmer/minimizer covers on average. This would give a clue as to how the minimizers and syncmers are distributed together with Figure 3.2. The Calculation is done by dividing the number of base pairs covered by the k-mers (excluding overlaps) by the number of k-mers themselves, The results are shown in Figure 3.3. Here the results are again skewed in the syncmers favor, and the difference between the medians of the two techniques increases in pace with the error rate, as does the spread of the results.

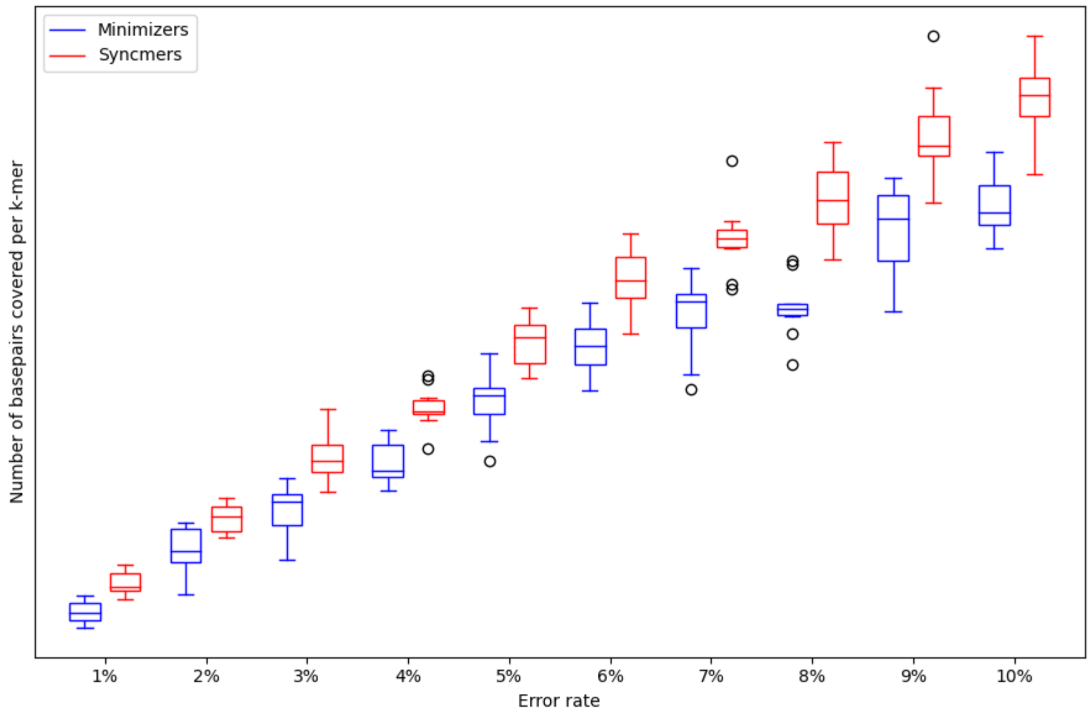

Figure 3.3: **Number of base pairs covered per conserved k-mer**

3.3 isONcorrect interval score

For this analysis, the results of the isONcorrect interval score is especially interesting, because the score is an important part of the algorithm directly used by isONcorrect to correct sequence reads. isONcorrect corrects reads by finding shared subsequences between reads. A subsequence is in this context a substring produced by two anchoring k-mers, the first and last k-mer of the subsequence. The subsequence is said to be shared with another read if the same start and end k-mer is found. The isONclust algorithm finds the combination of non-overlapping subsequences across the read that (1) best covers the read and (2) are shared between as many reads as possible. It does this through a scoring system which gives scores based on subsequence length, which is then given a multiplier for how many times it is found in other reads. By having a more conserved selection technique, we hypothesize that more intervals will be shared under errors, leading to higher score and better correction. The algorithm will run on 50 simulated reads per error rate.

The interval score spread is shown in Figure 3.4. The figure shows that the interval scores for syncmers are higher than the minimizers throughout the error rates. Although the syncmers scored higher across the board, the difference between minimizer and syncmer scores steadily increases at higher error rates.

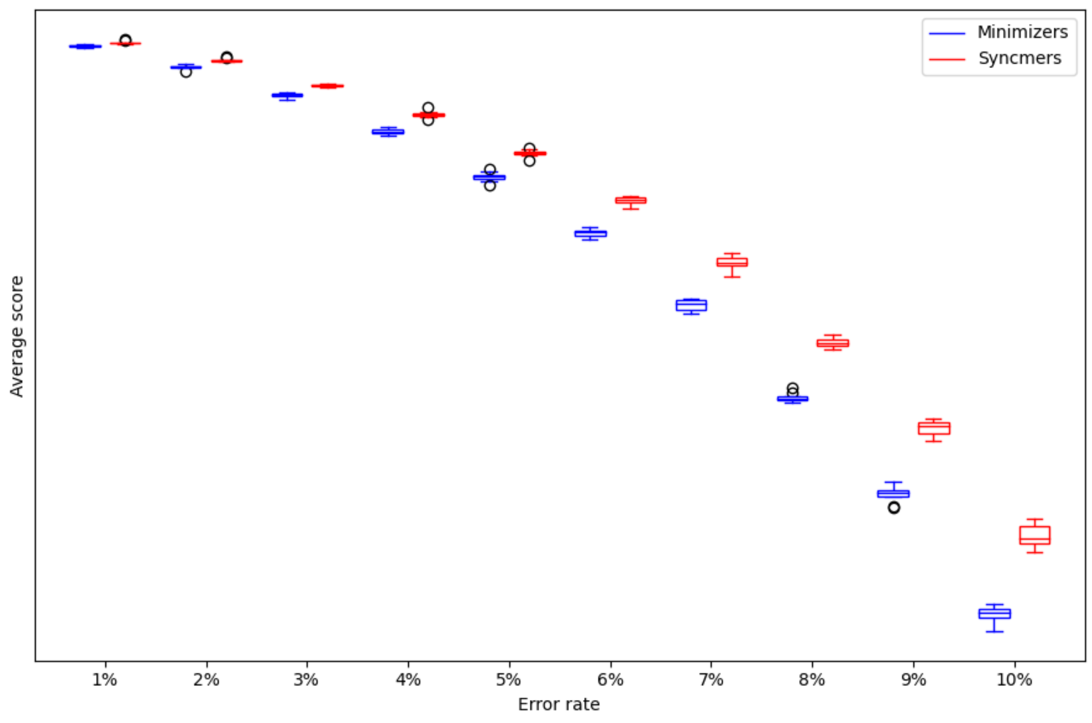

Figure 3.4: **The isONcorrect Interval score.** Higher score is given for long k-mer subsequences that are conserved across many reads. These are results based on 50 reads.

4 Discussion

4.1 Implementation of the open syncmer algorithm

An algorithm which generates open syncmers from a given sequence was successfully implemented in Python. The runtime is in $\mathcal{O}(n)$, but still needs to be looked over for optimization purposes. There is at least one bottleneck that has been identified that could be improved upon in the future. In the pseudocode shown in the methods section, the bottleneck was identified to be at the equivalent of line 5, where a comparison is made between the parameter t and argmin, the index of the minimal entry, of the queue. In the actual Python implementation the queue data structure that was used needed to be converted to a list in every iteration to be able to find argmin. To improve on this, perhaps some queue structure which allows for easier searching might be used, or just a list as a sort of pseudo-queue. Neither of these proposed improvements have been tested as the focus of this thesis lies on the performance comparison of minimizers and syncmers.

4.2 Results of measurements

As can be seen in figures 3.1 to 3.4, the results of the metrics measured so far look very promising for syncmers. In figures 3.2 to 3.4 the bulk of syncmer results clearly outscore the minimizers at every error rate, but it is not as clear cut in figure 3.1.

Figure 3.1 shows the conservation of k-mers between the two methods, which is the proportion of the original k-mers which are still selected when errors are introduced into the sequence. In his paper, Edgar claims a

4% improvement in conservation, namely 0.312 for syncmers vs 0.301 for minimizers at 10% error rate [1]. Although these numbers fall within the range of the results shown in figure 3.1, it again is not clear cut. The outliers show that minimizers in fact performed better than syncmers in several cases. Perhaps it could be that the total average would show a much similar result to Edgars, or perhaps it is a result of Edgar using real bacterial genome assemblies, while these figures come from tests run on script generated sequences. Either way it would be pertinent to find out when and why the syncmers fall short of the minimizers.

Figure 3.2 shows instead how much of the sequence, counted in base pairs, is covered by the k-mers conserved at different error rates. Though syncmers appear to more clearly outperform minimizers here, there are still some interesting outliers where minimizers suddenly perform unexpectedly well. As in figure 3.1, the circumstances which lead to those types of anomalies are currently unknown and could be worth looking into.

Looking at figure 3.2 with figure 3.1 still in mind, it is also interesting to notice that, although based on similar data of k-mer conservation, the base pairs covered per syncmer are higher than the number of base pairs covered per minimizer for all error rates. The reason for this might be that syncmers are generally more spread out across the sequence than minimizers are, which the results of figure 3.3 shines a light on.

Figure 3.3 illustrates the average number of base pairs covered per k-mer. Here, the syncmers are dominating, which could explain the results of figure 3.2. The syncmer k-mers on average do not seem to overlap as much as minimizers do, so they are more spread out across the sequence and cover a larger amount of base pairs than minimizers.

The difference between the minimizer and syncmer results appears to increase with higher error rates. It could be that minimizers have a tendency to conserve overlapping k-mers more than syncmers do.

Lastly, figure 3.4 shows the interval score calculated by isONcorrect. That the difference between syncmer and minimizer scores increases with error rate in a similar way as in figure 3.3, could indicate a possible relation between number of base pairs covered per k-mer and interval score.

4.3 Differences in characteristics of syncmers and minimizers

All these results together give rise to some possible conclusions about the differences between minimizers and syncmers. Minimizers are almost as effective as syncmers at conserving k-mers, but tend to have more overlap in k-mers. As errors are introduced, minimizers also appear to be more likely to conserve k-mers with more overlap than syncmers do. It could be that the syncmers are outperforming minimizers by consistently maintaining a larger spread of its k-mers. The scoring system of isONcorrect is designed to find subsequences that best cover a sequence read and are shared between as many reads as possible. The propensity of syncmers to spread out could make it more likely to conserve longer subsequences, as fewer k-mers can be used to cover a larger portion of base pairs.

4.4 Pitfalls and future work

Its important to note that all the data in the results come from sequences generated by a script, and that all the errors were induced randomly. Trying the same things on real sequences with real mutations or error reads might yield different results, which could be interesting to see. There were even sequences where minimizers appeared to perform better than syncmers, so figuring out what type of sequence this happens in would also be of interest.

It is not immediately apparent why minimizers would have a higher propensity than syncmers to conserve overlapping k-mers, as figure 3.3 could be hinting at. Trying to find out why this happens might also be an interesting lead.

Another thing that could be investigated is the difference in minimizer and syncmer scores increasing with error rate in figures 3.3 and 3.4. It might be the compounding issue of minimizers losing more of its non-overlapping k-mers with each step, but other factors may also be at play.

One thing that was not brought up in the results or main discussion is the drastic effect that the choice of parameters s and t have on the resulting syncmers. Changing either one of them will yield a very different output, so figuring out exactly how they work and the optimal way to set them for their intended use in specific applications would be very helpful.

An important first step for future work however, might be to optimize the code for the syncmer algorithm. When larger and more complicated datasets are to be explored and tested on, it will become increasingly important that the code performs efficiently.

All in all this small study does show good potential for syncmer based selection methods to continue building upon. Of course, before minimizers can safely be replaced in many implementations much further testing must be done to figure out all the kinks and potential problems that syncmers might have, as well as situations where minimizers may have an advantage.

5 References

[1] R. Edgar. "Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences". In: *PeerJ* 9 (2021). DOI: 10.7717/peerj.10805.

[2] H. Li. "Minimap2: pairwise alignment for nucleotide sequences". In: *Bioinformatics* 34.18 (2018). DOI: 10.1093/bioinformatics/bty191.

[3] M. Roberts et al. "Reducing storage requirements for biological sequence comparison". In: *Bioinformatics* 20.18 (2004). DOI: 10.1093/bioinformatics/bth408.

[4] K. Sahlin and P. Medvedev. "Error correction enables use of Oxford Nanopore technology for reference-free transcriptome analysis". In: *Nature Communications* 12.1 (2021). DOI: 10.1038/s41467-020-20340-8.

[5] S. Schleimer, D. Wilkerson, and A. Aiken. "Winnowing". In: *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03* (2003). DOI: 10.1145/872757.872770.

[6] J. Shaw and Y. W. Yu. "Theory of local k-mer selection with applications to long-read alignment". In: *BioRxiv* (2021). DOI: 10.1101/2021.05.22.445262.