DISTRIBUTED DE BRUIJN GRAPH USING MINIMIZERS

GLENN TESLER, ...

1. Introduction

Describe de Bruijn graph.

Describe condensed graph.

Short overview of de Bruijn assembly, noting that there are more steps but that constructing the full de Bruijn graph is a necessary step.

Describe strategy of assigning k-mers to different processors for initial phase of construction, then later chaining them together for other phases.

This paper focuses on assigning k-mers to different processors for the initial phase of construction. The goal is to do so in a way that roughly divides the time and memory evenly among the processors, and that consecutively overlapping k-mers (that chain together to form longer strings, as parts of contigs) should often be on the same processor.

2. A toy problem: minimum of multiple rolls of a die

Consider rolling a fair r-sided die (faces $0, \ldots, r-1$) w times and taking the value of the smallest roll. It is well-known from order statistics that

$$P(\text{minimum of } w \text{ rolls} = x) = (1 - x/r)^w - (1 - (x+1)/r)^w \qquad \text{for } x = 0, \ldots, r-1.$$

This generalizes easily to a biased r-sided die. Let p_x be the probability of rolling face x. The survival function is

$$Q(x) = P(X \geq x) = p_x + p_{x+1} + \cdots + p_{r-1} \qquad Q(r) = 0$$

and the probability that the minimum of w rolls is x is

$$P(\text{minimum of } w \text{ rolls} = x) = Q(x)^w - Q(x+1)^w \qquad \text{for } x = 0, \ldots, r-1. \tag{1}$$

3. Minimizers for sequence data

3.1. **Yorke.** Describe Yorke results.

Our notation (not the same as Yorke's various notations):

k = vertex size in de Bruijn graph

m = minimizer size

w = number of m-mer windows within k-mers = $k - m + 1$

[We'll use $k = 55$, $m = 8$, $w = 48$.]

Consider a k-mer $\tau = \tau_1 \tau_2 \ldots \tau_k$.

Let $\tau_{i:j}$ denote the substring $\tau_i \cdots \tau_j$.

For double-stranded sequences, let τ' denote the dual (reverse complement for basespace or reverse for colorspace). Let τ^* denote the *canonical word*, which is the lesser of τ or τ'. For now we will order strings lexicographically, but later we will consider other orderings (see Section 5).

For a single-stranded sequence, the m-minimizer of a k-mer is the smallest m-mer out of all w m-mer windows in the k-mer:

$$\text{minimizer}(\tau) = \min\left(\tau_{1:m},\ \cdots,\ \tau_{k-m+1:k}\right) \tag{2}$$

Date: December 26, 2010.

For a double-stranded sequence, we replace each m-mer by its canonical word:

$$\text{minimizer}(\tau) = \min\big((\tau_{1:m})^*, \cdots, (\tau_{k-m+1:k})^*\big) \tag{3}$$

We approximate the distribution of this by the die model by considering a biased die with a reduced number of sides, where we combine the faces for each dual pair of words x, x' together into one face. Although overlapping m-mer windows within a k-mer have dependencies, equation (1) still works well; see Figure 1. Also see the **Cumulative minimizer usage** curves attached at the end.

Yorke shows that the probability two consecutive k-mer windows have different minimizers is roughly $2/(w+1)$, but this is only an approximate result. For the die rolling model, it is actually

$$2\sum_{x=0}^{r-1}(Q(x)^w - Q(x+1)^w)(1-Q(x)) \ .$$

For the sequence minimizer problem, there are additional complications due to dependence of overlapping m-mers, and due to m-mers that may only be minimizers at ends but not in the interior.

3.2. **Balanced assignment of k-mers to n processors.** n = number of processors

We will create a table processor(M) that assigns each m-mer M to a processor $0, \ldots, n-1$.
For each k-mer u, we compute its minimizer M, and assign the k-mer u to processor(M).
If this assignment is sufficiently balanced, we obtain that for consecutive k-mer windows:

$$\begin{aligned} P(\text{change processor}) &= P(\text{change minimizer}) \cdot P(\text{change processor}|\text{change minimizer}) \\ &= \frac{2}{w+1} \cdot \left(1 - \frac{1}{n}\right) \end{aligned} \tag{4}$$

A uniformly distributed assignment of each k-mer to a number $0, \ldots, n-1$ would give a probability $1 - \frac{1}{n}$ to switch processors between consecutive k-mers.

With Hamid's current method of taking the lesser of its two $(k-1)$-mers, hashing it, and taking the hash modulo n, it is effectively $w = 2$, so the probability to change minimizers in consecutive k-mers is roughly $2/3$ and the probability to change processors is $(2/3)(1 - 1/n)$.

But with this approach, we could have, e.g., $k = 55$, $m = 8$, $w = k - m + 1 = 48$, so the probability to change minimizers is $(2/49)$ and the probability to change processors is $(2/49)(1 - 1/n)$, which is a $16\frac{1}{3}$-fold improvement. (This depends on k and m.)

3.3. **Balanced assignment of minimizers to processors.** m-mers that are smallest will tend to be minimizers for a greater number of k-mers.

We will assign all canonical m-mers to processors in a manner that roughly balances the number of k-mers (with multiplicities) assigned to each processor.

Constraints/goals:

(a) $1 \le m \le k$
(b) The table processor(M) is stored in memory on each processor, and has 4^m entries. To keep it small, say $1 \le m \le 12$. (In practice, many fewer m-mers actually occur as minimizers.)
(c) All m-mers should have probability of occurrence $\ll 1/n$. Additionally, we will be assigning multiple m-mer minmizers to the same processor, and adding up their probabilities to approximate $1/n$. If the probabilities of individual m-mers are too high, we will not be able to do this, so we will have to increase m. This condition imposes a lower bound on m.
(d) We want w as large as possible since the probability that adjacent k-mers have different minimizers is $2/(w+1)$. Increasing w gives decreasing m. However, conditions (b,c) are more important.

Training phase

Use training data (a reference genome, or a sample a subset of the reads) to determine one of the following:

- Directly generate a table of frequencies of every m-mer minimizer over all k-mers in training data.
  Let t_x be the empirical probability that m-mer x is a minimizer (the fraction of observed k-mers in training data for which it is the minimizer).

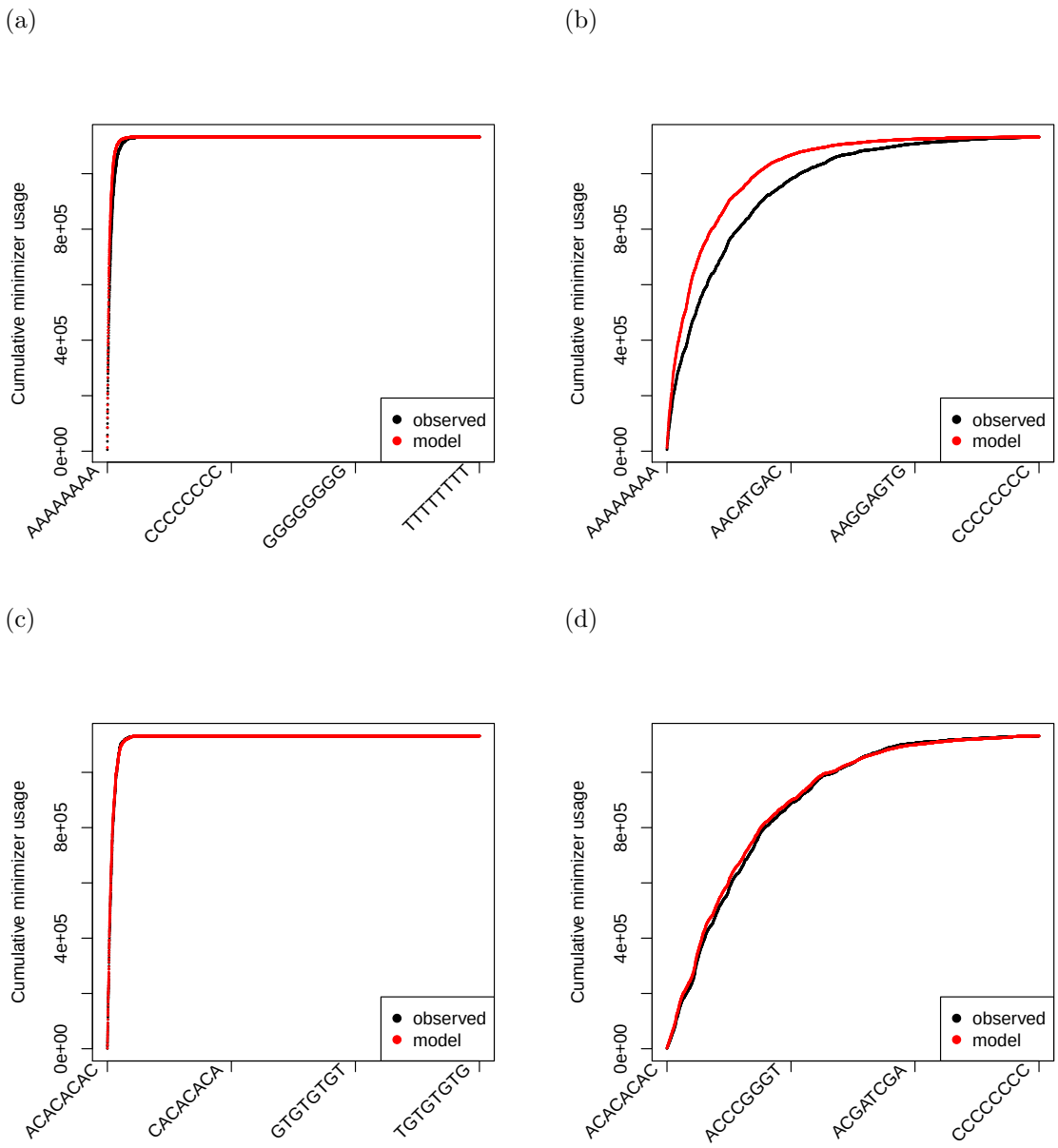


FIGURE 1. (a) Consider a die with all m-mers ($m = 8$), with dual m-mers combined together onto the same face, and the probability of each face based on m-mers in data. The red curve shows the cumulative distribution of minimizers for rolling this die $w = 48$ times, (derivable from equation (1)). The black curve shows the actual cumulative distribution of sequence minimizers (Eq. (3)) over all k-mers ($k = 55$) in the same data. (b) We "zoom in" on (a) in a nonlinear fashion: on the x-axis, we only show the m-mers that actually occur as minimizers. (c,d) Instead of alphabetical ordering with AAAAAAAA least, we use an ordering $<_s$ with $s =$ ACACACAC least; see Section 5. This breaks dependencies between adjacent m-mers in the sequence; spreads out the distribution a little more; and improves agreement between the red and black curves.

- Or, determine the frequencies p_x of every m-mer x in training data. Use (1) to estimate t_x from this.

We prefer to train from the empirical minimizer frequencies, but we will show that the empirical m-mer frequencies give a good estimate of minimizer frequencies at least for the most dominant minimizers.

Partition the m-mers into n groups where the probabilities of the minimizers in each group add up to approximately $1/n$.

Greedy algorithm: [very rough draft, will be rewritten]

For each processor $i = 0, \ldots, n-1$, initialize a "discrepancy" to $d_i = -1/n$.

Loop through all canonical m-mers x from least to greatest, skipping the ones whose minimizer probability is zero.

If there is at least one processor for which $d_i + t_x \leq 0$, then assign x to the processor that minimizes $|d_i + t_x|$. Otherwise (meaning $d_i + t_x > 0$ for all i), choose the processor i minimizing $|d_i + t_x|$.

Assign processor(x) $\leftarrow i$ and update $d_i \leftarrow d_i + t_x$.

NOTE: We can prove that there exists an assignment of minimizers to processors such that for all i, if $d_i > 0$ then removing the m-mer with smallest t_x gives $d_i - t_x \leq 0$. ... So as long as the discrepancy doesn't turn positive until high x (where t_x is small), all processors will have a small total discrepancy. If this is infeasible, redo the algorithm with a larger value of m.

Zero frequency minimizers in training data: For the m-mers that have zero (or low) probability to be minimizers in the training data: assign them in round-robin fashion equally among all n processors.

3.4. **End minimizers.** Zero and low frequency minimizers in the training data conceivably may be due to poor sampling. But usually they are due to the die model not adequately representing minimizers in a DNA sequence. E.g., for standard order with $m = 8$, the sequence CCCCCCAC is quite rare as a minimizer. To be the minimizer, it would have to appear as the final m-mer (and/or GTGGGGGG would have to appear as the first m-mer) of a k-mer that otherwise consists solely of C's and G's, e.g., $C^{53}AC$. Note that if this m-mer were in the middle of a sequence, the minimizer would have the form ACxxxxxx instead.

The Yorke paper discusses dealing with "end minimizers" and suggests using a hybrid procedure that involves using a certain value of m away from the ends, and smaller values of m near ends. We could look into this in the future, but haven't considered it yet. Note: The minimizer graph may provide a different workaround for the problem.

4. Double-strandedness

The minimizer is the minimum of the w canonical m-mers in a k-mer (see Eq. (3)).

In the die model, the effective number of faces on the die is reduced as follows:

- Single stranded: $r = 4^m$
- Double stranded basespace: $r = \begin{cases} (4^m)/2 & \text{if } m \text{ is odd,} \\ (4^m + 4^{m/2})/2 & \text{if } m \text{ is even.} \end{cases}$
- Double stranded colorspace: $r = \begin{cases} (4^m + 4^{(m-1)/2})/2 & \text{if } m \text{ is odd,} \\ (4^m + 4^{m/2})/2 & \text{if } m \text{ is even.} \end{cases}$

5. Orderings

The minimizer procedure works with any total ordering of the 4^m m-mers.

We define an ordering $<_s$ that makes a given word $s = s_1 \ldots s_m$ the smallest m-mer.

Let $x = x_1 \ldots x_m$ be an m-mer.

The binary representation $B(x)$ of an m-mer $x = x_1 \ldots x_m$ is to assign $A \to 00$, $C \to 01$, $G \to 10$, $T \to 11$ to each letter, with x_1 the most significant pair of bits and x_m the least significant pair.

For colorspace, convert base 4 to base 2: $0 \to 00$, $1 \to 01$, $2 \to 10$, $3 \to 11$.

We define a new total ordering on all m-mers as follows.

Let s be an arbitary m-mer. We define the ordering $<_s$ by

$$x <_s y \quad \text{iff} \quad B(x) \text{ xor } B(s) < B(y) \text{ xor } B(s)$$

(where on the right, xor is performed bitwise and $<$ is standard numerical order).

Note that in this ordering, s is the smallest m-mer. For each prefix $\sigma = s_1 \dots s_p$ of s, the m-mers starting with σ are smaller than m-mers starting with all other prefixes.

The Yorke papers describe different orderings of all 4^m m-mers. They observe that in the standard order, simple strings like AAAAAAAA may have high frequency, so an excessive number of strings will have it as a minimizer. The recommend reordering m-mers with more complex sequences as the minimum to avoid this. They are also concerned about GC-bias.

The ordering $<_s$ allows addressing the problem of homopolmer A's. More importantly, it allows a strategy to quickly scan a k-mer for minimizers.

6. Strategies for scanning for minimizers

We consider different strategies for scanning for k-mers.

If we are scanning a chain of consecutively overlapping k-mers (based on scanning a read, or on following edges in the de Bruijn graph), we use the following incremental strategy:

Incremental strategy

- The previous k-mer was $\tau_1 \dots \tau_k$ with minimizer $M = (\tau_{i:i+m-1})^*$, and now we are considering the k-mer $(\tau_{2:k+1})^*$.
- If $(\tau_{k+2-m:k+1})^* \leq M$ then output that the new minimizer is $(\tau_{k+2-m:k+1})^*$.
- If $i > 1$ then output that the new minimizer is still M.
- Since M has rolled off the other end of the k-mer, we must rescan the whole k-mer for the minimizer. We may use the "fast scan strategy" below.

Note: The experimental results include three variations of the incremental strategy.

- **"list" strategy:** When processing reads with arbitrary length, we roll an m-mer window through the entire length of the read, replacing the minimizer by the current m-mer window if the canonical form of the current window is $\leq_s$ the previous minimizer. The m-mer at positions $i, \dots, i+m-1$ is the last m-mer of the k-mer at positions $i+m-k, \dots, i+m-1$. If the minimizer rolls out of the start of the k-mer window, we recompute the minimizer by rescanning the last w m-mers. The Yorke papers have a version of this.
- **"right"/"left" strategy:** For processing two consecutive k-mers on an edge of the de Bruijn graph, we use the incremental strategy by rolling in a new m-mer from the "right" or the "left" depending on the direction. If the minimizer rolls out of the opposite end of the k-mer window, we must recompute it by rescanning the whole k-mer; we may do this with the "fast scan strategy" below.

Note: It will be straightforward to adapt the left and right strategies to the bidirected graph, but the direction will keep changing in the bidirected structure.

When following a 1-in/1-out chain of k-mers that reside on the same processor, you could keep track of the minimizer position for the current k-mer in a local variable. In case of ties, it is best to point at the most recent instance encountered, since it will take longest to roll out the other end.

Or, you could add a minimizer position field to the k-mer structure:

- If there is at least one instance of the minimizer away from the ends, point at any such instance. It really doesn't matter which instance you use, as long as it is away from the ends.
- If there only exists an instance of the minimizer at one end and not the other end or the interior, point at that one such instance.
- If there exists an instance of it at both ends but not in the interior, have a special position code indicating it is at both ends.

Fast scan strategy

- Initialization: choose a prefix size p ($1 \leq p \leq m$). Let $\sigma = s_1 \dots s_p$ where s is the mask used to reorder the m-mers.

  Precompute a table indexed by all m-mers x, with the following information:

  Does x start with prefix $s_1 \dots s_p$ (and/or its dual)?

  Does x contain another instance of this prefix and/or its dual (just the first one past the start) is sufficient? If not, does the suffix of x overlap with this prefix and/or its dual?

  - Does $x_1 x_2 \ldots x_p = \sigma$ (and/or its dual in the double-stranded case)?
    Store a flag dir equal to "forward", "reverse", "both", "notfound".
  - What is the least position jump (abbreviated j), if any, with $1 \leq j \leq m - p + 1$, for which $x_j \cdots x_{j+p-1} = \sigma$ (or its dual)?
  - Or if there is no such position, what is the least position $m - p + 2 \leq j \leq m$, for which $x_j \cdot x_m = s_1 s_2 \ldots s_{m-j+1}$ (or its dual)?
  - If there is still no such j, set $j = m$.
- Scanning k-mer $\tau_{1:k}$:
- Set $i = 1$.
- Set best $= \infty$ and bestpos $= \infty$.
- while $(i \leq k)$
- Let $x = \tau_{i:i+m-1}$
- Lookup dir and jump for x.
- If (dir == forward) or (dir == both), compare x against best. If $x \leq$ best then set best $= x$ and bestpos $= i$.
- If (dir == reverse) or (dir == both), let $y = (\tau_{i+p-m:i+p-1})'$. Compare y against best. If $y \leq$ best then set best $= y$ and bestpos $= i + p - m$.
- Set $i = i +$ jump.
- endwhile
- If best $== \infty$, use the full scan instead.
- Otherwise, return minimizer best at position bestpos.

Note: To avoid falling through to the full scan, we want high probability that we encounter σ at least once; but to reduce the number of m-mers we check in the scan, we would like the number of occurrences of σ to be small.

The expected number of occurrences of σ is roughly $w/4^p$ ($w \cdot \pi(\sigma)$ [to define — probability of sequence σ]) and we double that for double-stranded.

For $k = 55$, $m = 8$, $w = 48$, the value $p = 2$ is optimal. The expected number of occurrences of $\sigma = s_1 s_2$ is roughly 3 single stranded (6 double stranded), so most k-mers contain σ at least once, but not too many times.

The prefix $\sigma = CT$ is the fastest in dataset A [to expand].

It is also desireable to avoid overlaps of a suffix of σ with a prefix of σ or its dual, so prefixes $\sigma = AA$ or ATA will tend to be slower.

Basespace: Example: CTGTG... or CTGTG...A tend to be good

Colorspace: Example: 01212...123 tend to be good.

Full scan strategy

Scan all w canonical m-mers in the k-mer. Output the smallest one. If there are ties, use the one in the highest position.

Timing runs: Times are in seconds. The datasets are pretty small and there is some variability on rerunning it, so this will have to be redone more carefully.

Note that:

- For scanning strings of arbitrary length (as opposed to k-mers), the "list" mode (incremental strategy along whole string) is faster than any of the methods that break it into k-mers.
- For methods that break it into k-mers:
  - If we are following a chain of consecutively overlapping k-mers (including following edges of the de Bruijn graph), the left or right strategies are the best (and this can be adapted to following a chain in a bidirected graph).
  - If we are checking k-mers independently, the "fast" scan is better than the "full" scan.
- For prefixes:
  - The left/right strategies fall back on the "fast scan" mode with small probability (roughly $1/(w+1)$? chance that a minimizer rolls out the opposite end of a k-mer), so $p = 2$ still improve them a little.
  - The list and full strategies don't use the prefixes, so the variation in their timings is just random.
  - For $k = 55$, $m = 8$, double stranded, the mask CTGTGTGT with prefix size $p = 2$ tends to be slightly faster than AAAAAAAA, particularly for the "fast scan" strategy, although some timings are very close and it should be redone with a larger example.

Sample A, permutation mask AAAAAAAA

	list	left	right	fast	full
$p = 1$	0.31087	0.47207	0.46659	0.98297	0.97733
$p = 2$	0.31000	0.45022	0.44810	0.63171	0.97713
$p = 3$	0.31051	0.50536	0.45888	0.67231	0.97784
$p = 4$	0.31099	0.46906	0.46380	0.86702	0.97622

Sample A, permutation mask CTGTGTGT

	list	left	right	fast	full
$p = 1$	0.29333	0.47495	0.46770	1.00301	0.99676
$p = 2$	0.29349	0.45102	0.45084	0.58788	0.99396
$p = 3$	0.29330	0.45121	0.45271	0.61703	0.99089
$p = 4$	0.29349	0.47085	0.47054	0.91586	0.98019

Sample B, permutation mask AAAAAAAA

	list	left	right	fast	full
$p = 1$	0.69717	1.70554	1.57214	3.69080	3.79161
$p = 2$	0.69774	1.58889	1.54712	2.28179	3.82593
$p = 3$	0.70528	1.58870	1.56309	2.44561	3.72391
$p = 4$	0.70022	1.62216	1.69390	3.29755	3.74163

Sample B, permutation mask CTGTGTGT

	list	left	right	fast	full
$p = 1$	0.64671	1.69005	1.56515	3.76426	3.72197
$p = 2$	0.64529	1.56001	1.53069	2.12829	3.78496
$p = 3$	0.64619	1.57007	1.54055	2.18382	3.75090
$p = 4$	0.68485	1.61940	1.56472	3.57745	3.73166

Sample C, permutation mask AAAAAAAA

	list	left	right	fast	full
$p = 1$	0.46157	1.04084	1.00547	2.78676	2.31903
$p = 2$	0.44266	0.99121	0.95846	1.62290	2.37819
$p = 3$	0.43523	0.97693	0.95438	1.36322	2.33033
$p = 4$	0.43779	0.99168	0.97271	1.72432	2.32838

Sample C, permutation mask CTGTGTGT

	list	left	right	fast	full
$p = 1$	0.40598	0.99542	0.97010	1.81950	2.32189
$p = 2$	0.40409	0.97247	0.94852	1.30523	2.30738
$p = 3$	0.40350	0.99584	0.96466	1.73453	2.30712
$p = 4$	0.40218	1.01467	0.97022	2.26836	2.36368

7. Training

Implemented: Scan a a subset of reads, or a reference genome. Determine all its m-mer frequencies, or for all its minimizer frequencies in k-mer windows.

Potential future work: Take a reference genome. Compute a table of minimizer frequencies for the reference genome, augmented by minimizer frequencies in local regions upon making every possible SNP (one base mutation, insertion, or deletion).

8. Results

Postponed to end of document.

9. Increasing expected length on same processor

The algorithm is NOT YET IMPLEMENTED. It may or may not pan out.

The existing algorithm gives expected run lengths $\sim (w+1)/2$ for runs of consecutive k-mers with the same minimizer, and expected run lengths $\sim (w+1)/(2(1-\frac{1}{n}))$ for runs of consecutive k-mers on the same processor. We may be able to increase the expected size of runs of consecutive k-mers on the same processor, by assigning minimizers that tend to be conecutive to the same processor.

We define the *minimizer adjacency graph* as follows.

The vertices are all m-mers that occur as minimizers in the data.

The weight of vertex v is the number of times it occurs as a minimzer over all k-mers examined in the data.

The edges are undirected. $\{v, w\}$ is an edge if $v \neq w$ and consecutive k-mers in the data switch from minimizer v to w or vice-versa. The weight of an edge is the number of times this transition occurs in the data.

We want to assign minimizers to processors to optimize these things:

(1) Processor utilizations are roughly balanced.

The utilization of processor i is the sum of vertex weights of all vertices that are assigned to processor i.

We want the utlization of all processors to be close to $1/n$.

(2) A variation of a "cut minimization" problem, but we have n clusters instead of the traditional two (source component and sink component):

Minimize the sum of edge weights over all $\{v, w\}$, where v and w are assigned to different processors. This is the flow through the "cuts" between the processors.

Equivalently, maximize the sum of edge weights over all $\{v, w\}$ where $v \neq w$ are assigned to the same processor. This tends to increase the sizes of runs that stay on the same processor.

(3) Minimize training time. Condition (1) is more important than condition (2), and we already have an algorithm implemented to efficiently satisfy condition (1). If the training time to balance (1) and (2) is too long, the improvement in consecutive processor run lengths it gives may be irrelevant. So instead of minimizing cut flow, we can allow a suboptimal solution of (2) as long as its training time is fast and the cut flow is relatively small.

I have partial work on these graphs; it may or may not pan out.

10. Results

Datasets:

(A) First 25000 reads of E. coli lane 1 ($\approx 0.5\times$ coverage).
(B) E. coli reference
(C) S. aureus reference

We train on one of these, and evaluate on one of these. [Later, in addition to A, I will try other sample sizes like $1\times$, $10\times$, and/or all the reads. Hopefully it will turn out that it suffices to train on either on a reference genome (which is very small compared to a read file), or on a small sample of reads.]

Plain text statistics printouts: We would instead either put a few select numbers into the text, or would reformat some of it as tables.

Plots:

 # reads with a certain # runs: For evaluation on dataset A: the reads are 100 bases long and consist of 46 consecutively overlapping windows of 55-mers. However, some reads have N's, which effectively reduces this.

For evaluation on datasets B,C: The input is the single chromosome genome sequence, so it's effectively one long read. Instead of a plot, we would just report the number of runs rather than show the plot.

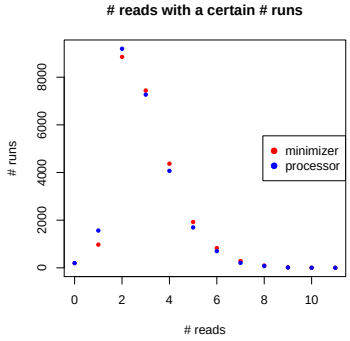
 # runs by run length histogram: Note the spike in runs whose length is exactly $w = k - m + 1 = 55 - 8 + 1 = 48$. The behavior of the curves change at multiples of w. In this example, the red curve shows there are some minimizer runs of length up to $2w = 96$ (when the minimizer accidentally stays the same in $2w$ consecutive positions). The blue curve shows there are a lot of processor runs of length up to $2w = 96$, and even some beyond this; this happens when the minimizers change along the string but are assigned to the same processor. *In progress:* The minimizer adjacency graph may provide a way to boost the length of processor runs by systematically assigning minimizers to the same processor when they are likely to be consecutive.

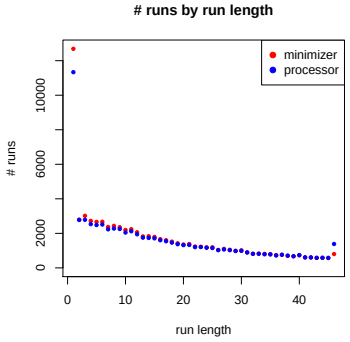
Graphics – cumulative minimizer usage: Previously described in the text.

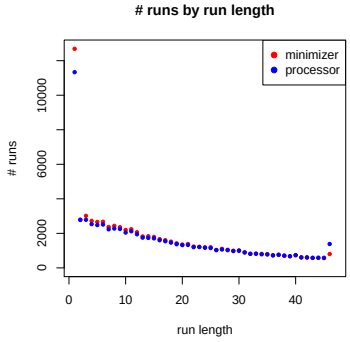
Train on (A), evaluate on (A), minimum s = AAAAAAAA

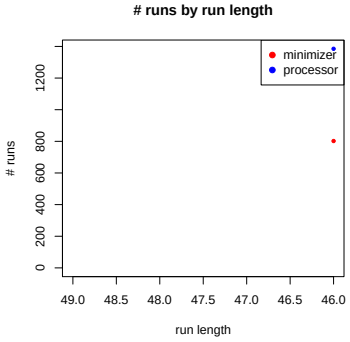
```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 1131857
# mmers (with repeats) in evaluation data = 2308330

# distinct minimizers observed = 3720
Highest minimizer observed = CCCCCCCC

# runs of same minimizer = 76127
# runs of same processor = 72995

average minimizer run length = 14.868000 +/- 12.744400   median = 12.000000   mode = 1.000000
average processor run length = 15.506000 +/- 13.094300   median = 13.000000   mode = 1.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0        70717 6.247874   4573         15.4640 13.0955
1        70716 6.247786   4586         15.4200 13.2887
2        70716 6.247786   4601         15.3697 12.9898
3        70719 6.248051   4422         15.9925 13.2709
4        71056 6.277825   4570         15.5484 13.2580
5        70717 6.247874   4589         15.4101 13.0090
6        70716 6.247786   4538         15.5831 13.2097
7        70717 6.247874   4484         15.7710 13.1032
8        70740 6.249906   4588         15.4185 12.9998
9        70717 6.247874   4525         15.6281 13.1765
10       70729 6.248934   4595         15.3926 12.9584
11       70723 6.248404   4590         15.4081 13.1720
12       70716 6.247786   4615         15.3231 12.8824
13       70716 6.247786   4556         15.5215 13.0504
14       70717 6.247874   4572         15.4674 13.0861
15       70725 6.248581   4591         15.4051 12.9664

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AAAAAAAA        5538    0.4892844    231         23.9740 14.0099
AAAAAAAC        3724    0.8183013    187         19.9144 14.5048
AAAAAAAG        3778    1.1520890    181         20.8729 14.2107
AAAAAAAT        6881    1.7600280    319         21.5705 14.5213
AAAAAACA        4641    2.1700621    240         19.3375 15.2266
AAAAAACC        4298    2.5497921    205         20.9659 14.5270
AAAAAACG        3484    2.8576048    155         22.4774 14.6887
AAAAAACT        2917    3.1153229    132         22.0985 14.4417
AAAAAAGA        3796    3.4507009    200         18.9800 15.3505
AAAAAAGC        4295    3.8301658    227         18.9207 14.6856

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AGGGGGGG           5     99.96925      1             5.0 0.00000
CAAGCCGA           1     99.96934      1             1.0 0.00000
CACCCCCC           1     99.96943      1             1.0 0.00000
CCACCCCC           1     99.96952      1             1.0 0.00000
```

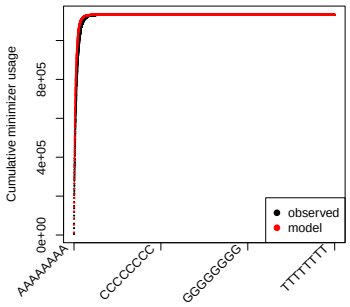
```
CCCACCCC           1     99.96961      1            1.0 0.00000
CCCCACCC           1     99.96970      1            1.0 0.00000
CCCCCACC           1     99.96978      1            1.0 0.00000
CCCCCCAC           1     99.96987      1            1.0 0.00000
CCCCCCCA           1     99.96996      1            1.0 0.00000
CCCCCCCC         340    100.00000      8           42.5 6.63325

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
           % kmers     % mmers
A------- 99.9692541 44.24228771
AA------ 99.6572005 14.57581888
AAA----- 81.6119881  4.91047641
AAAA---- 41.4364182  1.67458726
AAAAA--- 17.6879235  0.58891060
AAAAAA--  6.0208136  0.18931435
AAAAAAA-  1.7600280  0.06108312
AAAAAAAA  0.4892844  0.02473650
```

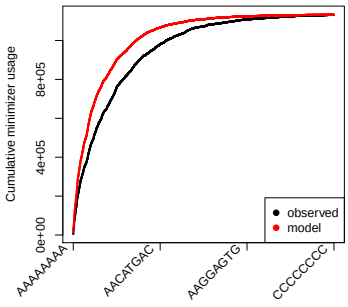








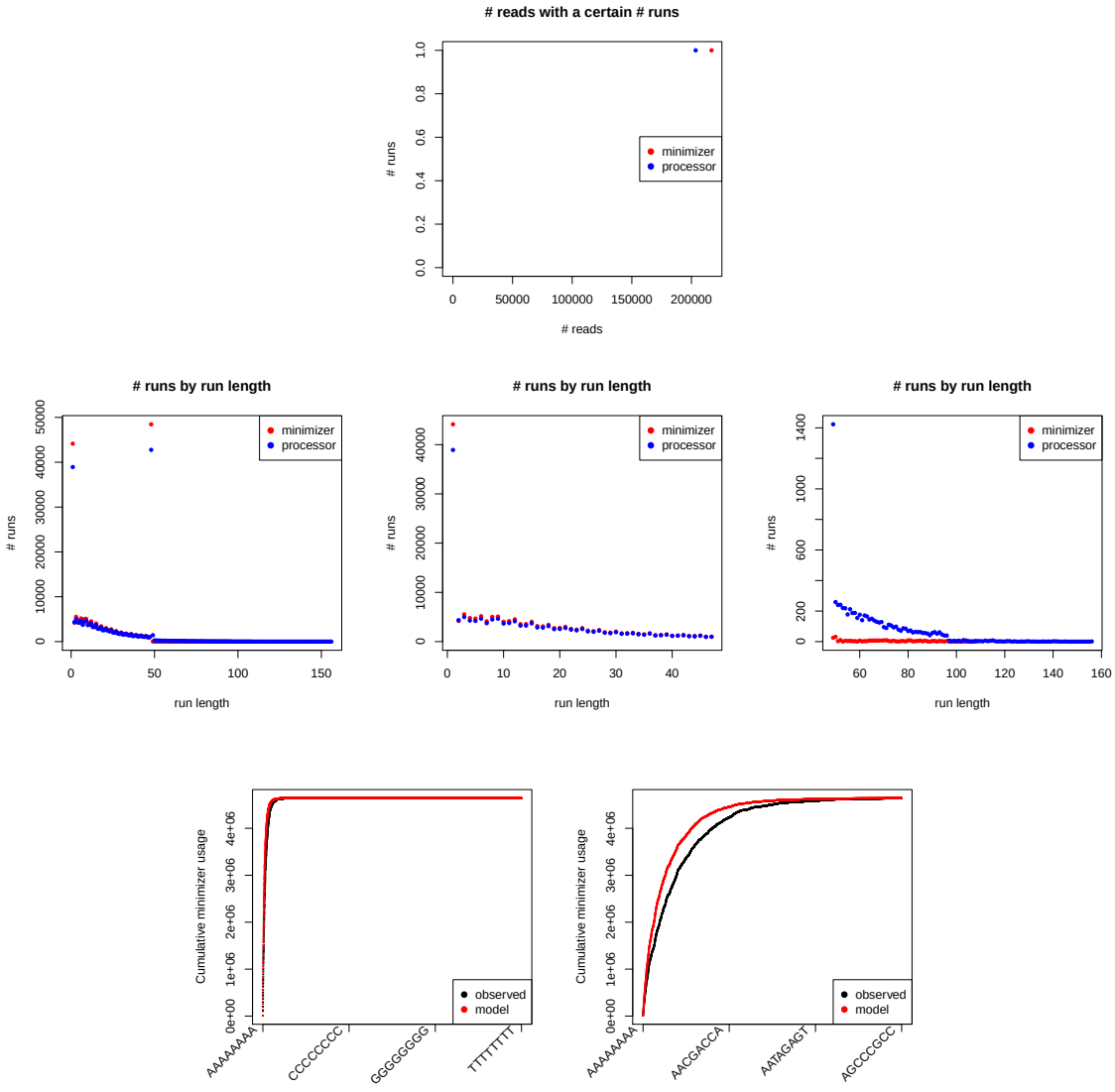




Train on (A), evaluate on (B), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 4639621
# mmers (with repeats) in evaluation data = 4639668

# distinct minimizers observed = 4619
Highest minimizer observed = AGCCCGCC

# runs of same minimizer = 216870
# runs of same processor = 203552

average minimizer run length = 21.393600 +/- 18.343300   median = 17.000000   mode = 48.000000
average processor run length = 22.793300 +/- 19.606200   median = 19.000000   mode = 48.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0       284528 6.132570  12558         22.6571 19.3461
1       298775 6.439642  13130         22.7551 19.6280
2       280066 6.036398  12477         22.4466 19.5709
3       283335 6.106857  12324         22.9905 19.6194
4       293682 6.329870  13052         22.5009 19.5378
5       293505 6.326056  12858         22.8266 19.7421
6       289632 6.242579  12660         22.8777 19.5467
7       292139 6.296613  12619         23.1507 19.6181
8       294968 6.357588  12898         22.8693 19.4656
9       297030 6.402032  12774         23.2527 19.9992
10      300572 6.478374  12871         23.3527 19.7777
11      299437 6.453911  13065         22.9190 19.6962
12      275490 5.937769  12126         22.7190 19.5627
13      292410 6.302454  12902         22.6639 19.7006
14      283564 6.111792  12750         22.2403 19.3825
15      280488 6.045494  12488         22.4606 19.4573

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length   std dev
AAAAAAAA       10770    0.2321310    224         48.0804  0.272454
AAAAAAAC       13499    0.5230815    348         38.7902 18.811800
AAAAAAAG       11529    0.7715716    314         36.7166 20.858400
AAAAAAAT       20102    1.2048398    497         40.4467 17.559400
AAAAAACA       14837    1.5246288    408         36.3652 20.345600
AAAAAACC       11959    1.7823870    358         33.4050 21.868900
AAAAAACG       13887    2.0817002    379         36.6412 20.067200
AAAAAACT       11413    2.3276901    281         40.6157 17.017900
AAAAAAGA       15503    2.6618338    421         36.8242 19.714400
AAAAAAGC       20233    3.0979255    546         37.0568 19.916000

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACTGAGCA           3     99.99950      1               3       0
ACTGAGGA           1     99.99953      1               1       0
ACTGATGC           3     99.99959      1               3       0
ACTGCCCG           4     99.99968      1               4       0
```

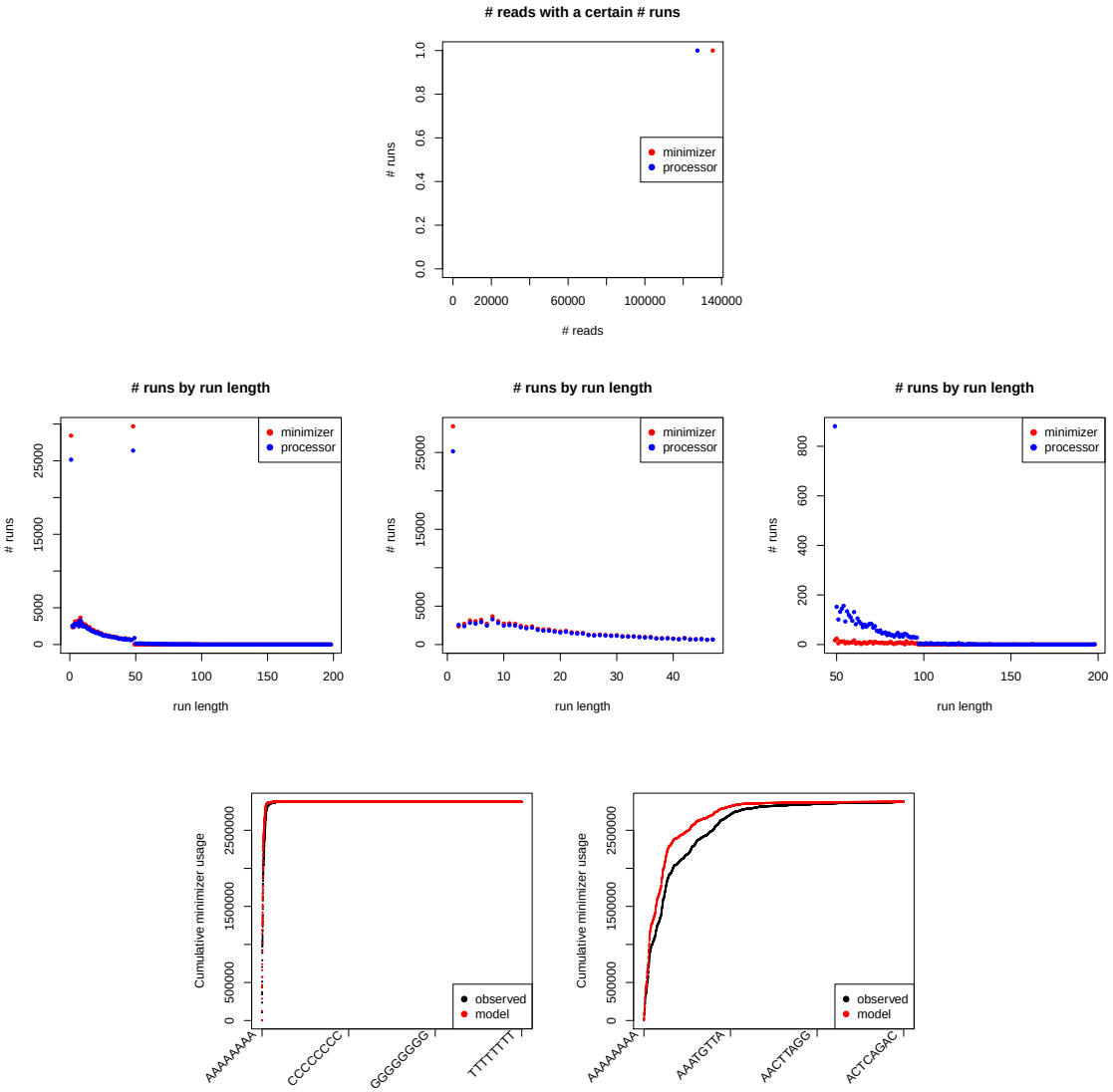
```
ACTGCGCG          2     99.99972      1              2       0
ACTGCGGC          1     99.99974      1              1       0
ACTGGCGC          2     99.99978      1              2       0
ACTTCATC          4     99.99987      1              4       0
AGATCCGC          3     99.99994      1              3       0
AGCCCGCC          3    100.00000      1              3       0

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
             % kmers     % mmers
A------- 100.0000000 43.08198345
AA------  99.7002988 14.04854830
AAA-----  81.0667509  4.67436463
AAAA----  39.4785695  1.52181147
AAAAA---  15.9173562  0.49846239
AAAAAA--   5.0087281  0.13798401
AAAAAAA-   1.2048398  0.03045477
AAAAAAAA   0.2321310  0.00521589
```



Train on (A), evaluate on (C), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 2872861
# mmers (with repeats) in evaluation data = 2872908

# distinct minimizers observed = 2858
Highest minimizer observed = ACTCAGAC

# runs of same minimizer = 135377
# runs of same processor = 127362

average minimizer run length = 21.221200 +/- 18.423200   median = 17.000000   mode = 48.000000
average processor run length = 22.556700 +/- 19.622800   median = 18.000000   mode = 48.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0       143957 5.010928   6816         21.1205 18.9496
1       180877 6.296058   8349         21.6645 19.0948
2       188753 6.570210   8352         22.5997 19.7806
3       149889 5.217412   6543         22.9083 19.5599
4       198450 6.907748   8823         22.4923 19.8264
5       146895 5.113196   7001         20.9820 18.5600
6       182138 6.339952   7894         23.0730 19.9547
7       181577 6.320424   8300         21.8767 19.1257
8       194899 6.784143   8467         23.0187 19.6979
9       176591 6.146869   7891         22.3788 19.6297
10      182667 6.358365   7998         22.8391 19.9253
11      199855 6.956654   8612         23.2066 20.0206
12      185330 6.451060   7892         23.4833 19.8685
13      169114 5.886606   7398         22.8594 19.6214
14      185999 6.474347   8149         22.8248 19.8974
15      205870 7.166027   8877         23.1914 19.9101

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length   std dev
AAAAAAAA        4566    0.1589356     95         48.0632  0.244537
AAAAAAAC       13696    0.6356729    310         44.1806 13.032200
AAAAAAAG       17216    1.2349362    389         44.2571 13.051100
AAAAAAAT       31666    2.3371823    699         45.3019 12.565100
AAAAAACA       18748    2.9897722    536         34.9776 21.101000
AAAAAACC        5450    3.1794786    141         38.6525 18.534900
AAAAAACG        6732    3.4138094    201         33.4925 21.152800
AAAAAACT        9651    3.7497463    283         34.1025 21.060000
AAAAAAGA       27126    4.6939619    752         36.0718 20.532700
AAAAAAGC       17624    5.3074270    443         39.7833 18.641200

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACACTGGT           6     99.99575      1               6 0.00000
ACAGACTG           1     99.99579      1               1 0.00000
ACAGATAC           9     99.99610      1               9 0.00000
ACAGCCCC          30     99.99715      5               6 0.00000
```

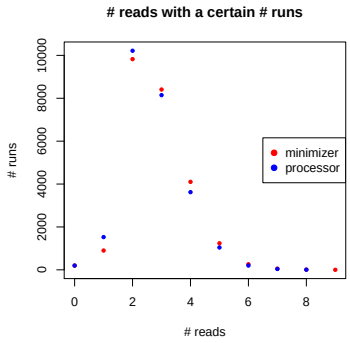
```
ACAGCGAC          36     99.99840      3             12 6.00000
ACAGGGGC           7     99.99864      1              7 0.00000
ACATAGGC           6     99.99885      1              6 0.00000
ACCAGCTG          14     99.99934      2              7 5.65685
ACGATATC           1     99.99937      1              1 0.00000
ACTCAGAC          18    100.00000      1             18 0.00000

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
             % kmers      % mmers
A------- 100.0000000 56.195534281
AA------  99.9954053 23.083335770
AAA-----  95.6739292  8.660701979
AAAA----  63.9027436  3.038976535
AAAAA---  29.9283188  1.005009558
AAAAAA--   9.9265158  0.270596900
AAAAAAA-   2.3371823  0.052908064
AAAAAAAA   0.1589356  0.003515602
```

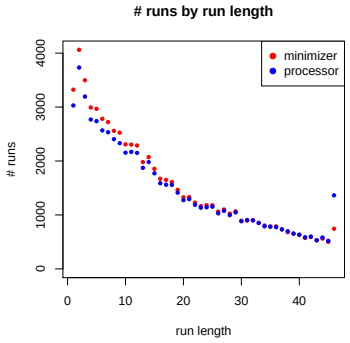


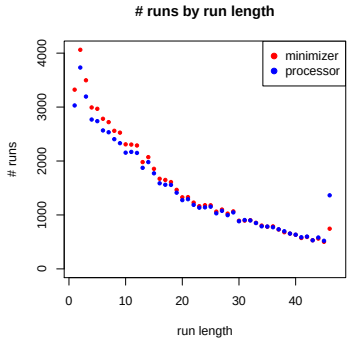
Train on (A), evaluate on (A), minimum s = ACACACAC

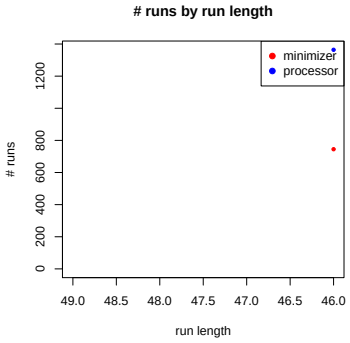
```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 1131857
# mmers (with repeats) in evaluation data = 2308330

# distinct minimizers observed = 3597
Highest minimizer observed = CCCCCCCC

# runs of same minimizer = 70419
# runs of same processor = 67638

average minimizer run length = 16.073200 +/- 12.259300   median = 14.000000   mode = 2.000000
average processor run length = 16.734000 +/- 12.645100   median = 15.000000   mode = 2.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0        70717 6.247874   4277         16.5343 12.4913
1        70717 6.247874   4261         16.5963 12.6053
2        70716 6.247786   4236         16.6941 12.6786
3        70716 6.247786   4140         17.0812 12.5039
4        70725 6.248581   4295         16.4668 12.5057
5        70722 6.248316   4204         16.8225 12.5685
6        70719 6.248051   4220         16.7581 12.6388
7        70723 6.248404   4182         16.9113 12.7665
8        71058 6.278002   4227         16.8105 12.6833
9        70723 6.248404   4303         16.4357 12.5199
10       70720 6.248139   4252         16.6322 12.6375
11       70717 6.247874   4106         17.2228 12.7498
12       70717 6.247874   4295         16.4650 12.8614
13       70720 6.248139   4172         16.9511 12.6846
14       70721 6.248227   4204         16.8223 12.6188
15       70726 6.248669   4264         16.5868 12.7953

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACACACAC         923    0.0815474     58         15.9138 12.8995
ACACACAA        1442    0.2089487     65         22.1846 14.4524
ACACACAT         844    0.2835164     40         21.1000 14.6336
ACACACAG         806    0.3547268     40         20.1500 13.9276
ACACACCC        1032    0.4459044     60         17.2000 13.3516
ACACACCA        1611    0.5882369     82         19.6463 14.2823
ACACACCT         666    0.6470782     34         19.5882 10.3488
ACACACCG        1417    0.7722707     72         19.6806 14.0460
ACACACGC         525    0.8186547     29         18.1034 10.9393
ACACACGA         464    0.8596492     21         22.0952 14.1099

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AATAATTA           5     99.96705      1            5.00 0.00000
AATTATCA           2     99.96722      1            2.00 0.00000
ATCCGGCG           4     99.96758      1            4.00 0.00000
ATCAGCGC           2     99.96775      1            2.00 0.00000
```

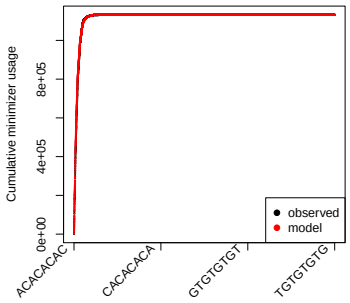
```
AGAGGGGC          12     99.96881      1          12.00 0.00000
AGGGGGGG           5     99.96925      1           5.00 0.00000
CCACCCCC           2     99.96943      1           2.00 0.00000
CCCCACCC           2     99.96961      1           2.00 0.00000
CCCCCCAC           2     99.96978      1           2.00 0.00000
CCCCCCCC         342    100.00000      8          42.75 6.20484

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
           % kmers      % mmers
A------- 99.9692541 44.242287714
AC------ 99.6919222 10.672434184
ACA----- 72.8240405  2.784047342
ACAC---- 22.2191496  0.578210221
ACACA---  5.6709461  0.139711393
ACACAC--  1.2738358  0.033833984
ACACACA-  0.3547268  0.010267163
ACACACAC  0.0815474  0.003552352
```

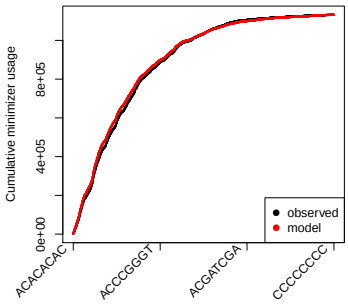








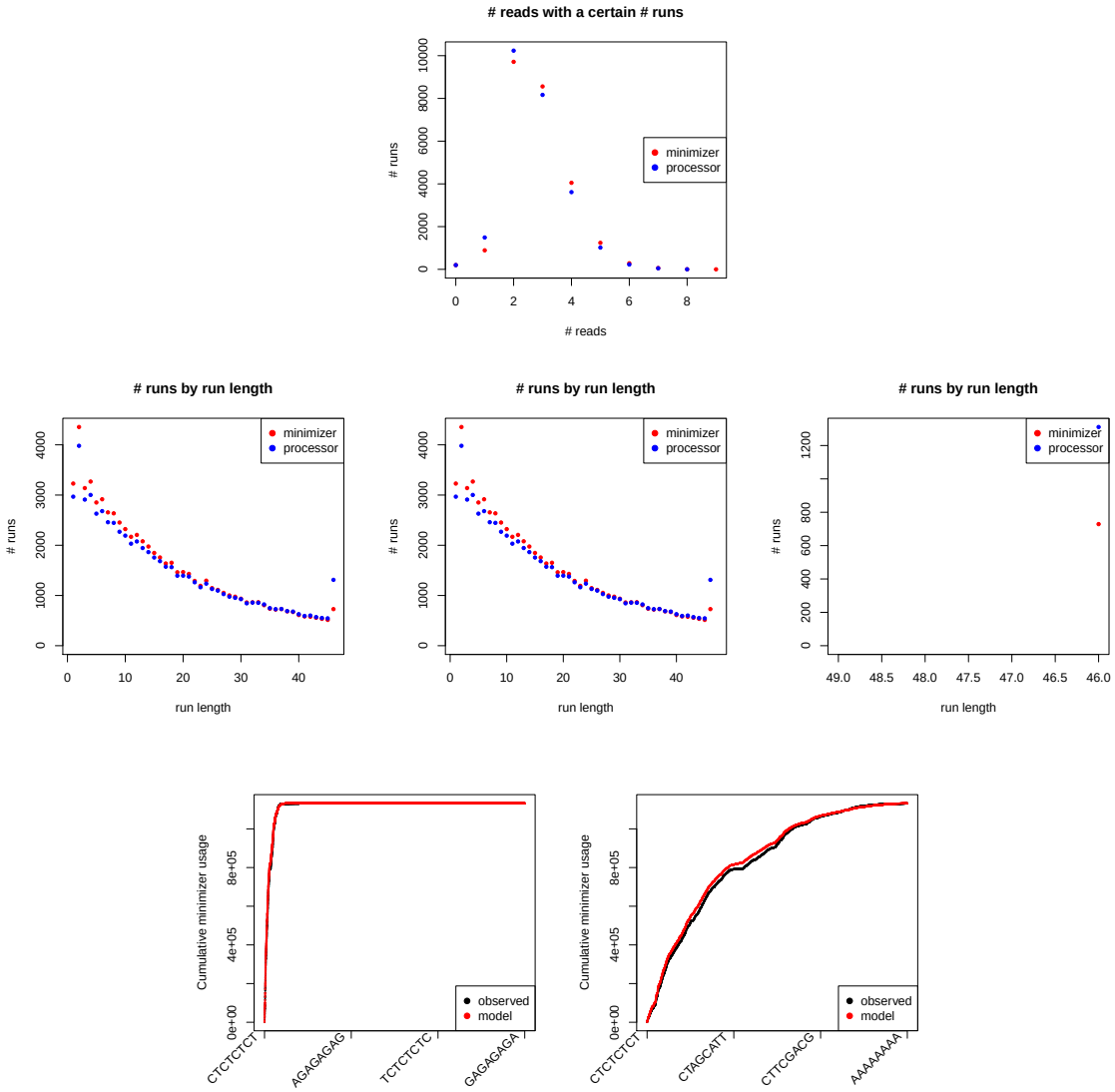




Train on (A), evaluate on (A), minimum s = CTCTCTCT

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 1131857
# mmers (with repeats) in evaluation data = 2308330

# distinct minimizers observed = 3968
Highest minimizer observed = AAAAAAAA

# runs of same minimizer = 70550
# runs of same processor = 67724

average minimizer run length = 16.043300 +/- 12.213900   median = 14.000000   mode = 2.000000
average processor run length = 16.712800 +/- 12.612600   median = 15.000000   mode = 2.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0        70705 6.246814   4270         16.5585 12.4226
1        70695 6.245930   4205         16.8121 12.6735
2        70724 6.248493   4179         16.9237 12.5558
3        70697 6.246107   4249         16.6385 12.6236
4        70709 6.247167   4119         17.1665 12.8054
5        70695 6.245930   4222         16.7444 12.6458
6        70694 6.245842   4283         16.5057 12.4956
7        70744 6.250260   4269         16.5716 12.5642
8        70726 6.248669   4345         16.2776 12.4719
9        70697 6.246107   4151         17.0313 12.7911
10       70698 6.246195   4184         16.8972 12.7208
11       70778 6.253263   4230         16.7324 12.5112
12       70724 6.248493   4213         16.7871 12.5209
13       70704 6.246726   4282         16.5119 12.6416
14       71135 6.284805   4276         16.6359 12.7350
15       70732 6.249199   4247         16.6546 12.6224

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
CTCTCTCT        1011   0.08932224     49         20.6327 14.2386
CTCTCTCG        1030   0.18032313     41         25.1220 13.3738
CTCTCTCC        1563   0.31841478     77         20.2987 12.8985
CTCTCTCA        1047   0.41091763     55         19.0364 12.1487
CTCTCTAT        1552   0.54803743     61         25.4426 14.1745
CTCTCTAG         114   0.55810937      6         19.0000 13.3716
CTCTCTAC         485   0.60095931     24         20.2083 12.4516
CTCTCTAA         243   0.62242845     11         22.0909 11.9453
CTCTCTTT        2577   0.85010739    122         21.1230 13.0666
CTCTCTTG         747   0.91610513     35         21.3429 13.2019

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
CAATATAA          37     99.98507      3         12.3333 1.15470
CAATAAAC           1     99.98516      1          1.0000 0.00000
CAATTTAA           4     99.98551      1          4.0000 0.00000
CAATGATG          22     99.98745      4          5.5000 2.38048
```

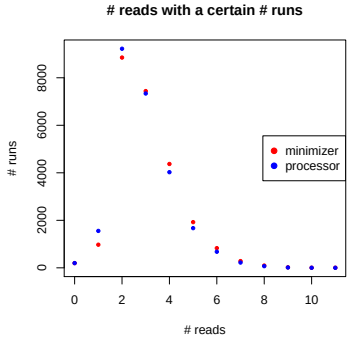
```
CAACATAT          7     99.98807      1          7.0000 0.00000
CATTATTT         36     99.99125      3         12.0000 7.93725
CATTTCAT          8     99.99196      2          4.0000 0.00000
ATTTATTT          2     99.99214      1          2.0000 0.00000
ATTTTTTT          4     99.99249      1          4.0000 0.00000
AAAAAAAA         85    100.00000      2         42.5000 4.94975

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
             % kmers      % mmers
C------- 99.99196012 42.898242452
CT------ 99.66974627 10.278166467
CTC----- 61.74525581  2.124869494
CTCT---- 21.92335251  0.571712017
CTCTC---  5.05911966  0.128231232
CTCTCT--  1.64004817  0.039032547
CTCTCTC-  0.41091763  0.009833949
CTCTCTCT  0.08932224  0.002122747
```

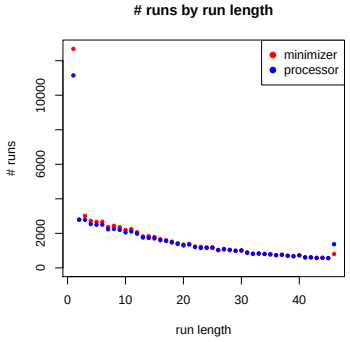


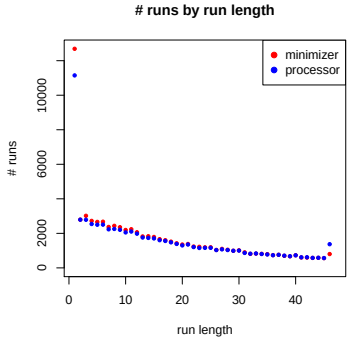
Train on (B), evaluate on (A), minimum s = AAAAAAAA

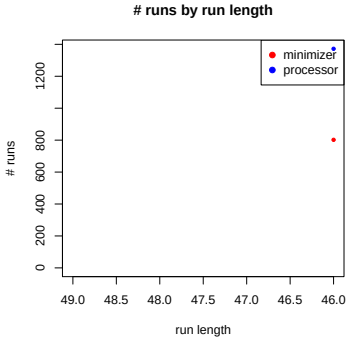
```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 1131857
# mmers (with repeats) in evaluation data = 2308330

# distinct minimizers observed = 3720
Highest minimizer observed = CCCCCCCC

# runs of same minimizer = 76127
# runs of same processor = 72835

average minimizer run length = 14.868000 +/- 12.744400   median = 12.000000   mode = 1.000000
average processor run length = 15.540000 +/- 13.080800   median = 13.000000   mode = 1.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0        75510 6.671337   4841         15.5980 13.1390
1        71908 6.353099   4676         15.3781 13.0129
2        69526 6.142649   4574         15.2003 13.0641
3        71360 6.304683   4556         15.6629 13.1438
4        70017 6.186029   4576         15.3009 13.1050
5        70033 6.187442   4392         15.9456 13.3811
6        72190 6.378014   4555         15.8485 13.2108
7        68841 6.082129   4496         15.3116 12.9661
8        68937 6.090610   4408         15.6391 12.9973
9        68428 6.045640   4456         15.3564 13.1177
10       70393 6.219249   4555         15.4540 12.8369
11       69879 6.173836   4594         15.2109 12.9670
12       73982 6.536338   4638         15.9513 13.2199
13       69168 6.111019   4454         15.5294 12.8398
14       69460 6.136818   4513         15.3911 13.0534
15       72225 6.381106   4551         15.8701 13.2079

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AAAAAAAA        5538    0.4892844    231         23.9740 14.0099
AAAAAAAC        3724    0.8183013    187         19.9144 14.5048
AAAAAAAG        3778    1.1520890    181         20.8729 14.2107
AAAAAAAT        6881    1.7600280    319         21.5705 14.5213
AAAAAACA        4641    2.1700621    240         19.3375 15.2266
AAAAAACC        4298    2.5497921    205         20.9659 14.5270
AAAAAACG        3484    2.8576048    155         22.4774 14.6887
AAAAAACT        2917    3.1153229    132         22.0985 14.4417
AAAAAAGA        3796    3.4507009    200         18.9800 15.3505
AAAAAAGC        4295    3.8301658    227         18.9207 14.6856

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AGGGGGGG           5     99.96925      1             5.0 0.00000
CAAGCCGA           1     99.96934      1             1.0 0.00000
CACCCCCC           1     99.96943      1             1.0 0.00000
CCACCCCC           1     99.96952      1             1.0 0.00000
```

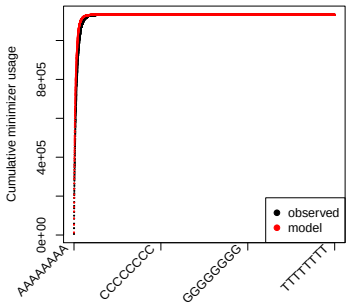
```
CCCACCCC           1     99.96961      1            1.0 0.00000
CCCCACCC           1     99.96970      1            1.0 0.00000
CCCCCACC           1     99.96978      1            1.0 0.00000
CCCCCCAC           1     99.96987      1            1.0 0.00000
CCCCCCCA           1     99.96996      1            1.0 0.00000
CCCCCCCC         340    100.00000      8           42.5 6.63325

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
           % kmers     % mmers
A------- 99.9692541 44.24228771
AA------ 99.6572005 14.57581888
AAA----- 81.6119881  4.91047641
AAAA---- 41.4364182  1.67458726
AAAAA--- 17.6879235  0.58891060
AAAAAA--  6.0208136  0.18931435
AAAAAAA-  1.7600280  0.06108312
AAAAAAAA  0.4892844  0.02473650
```

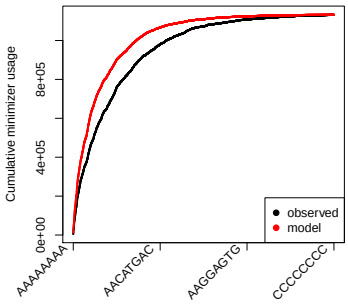








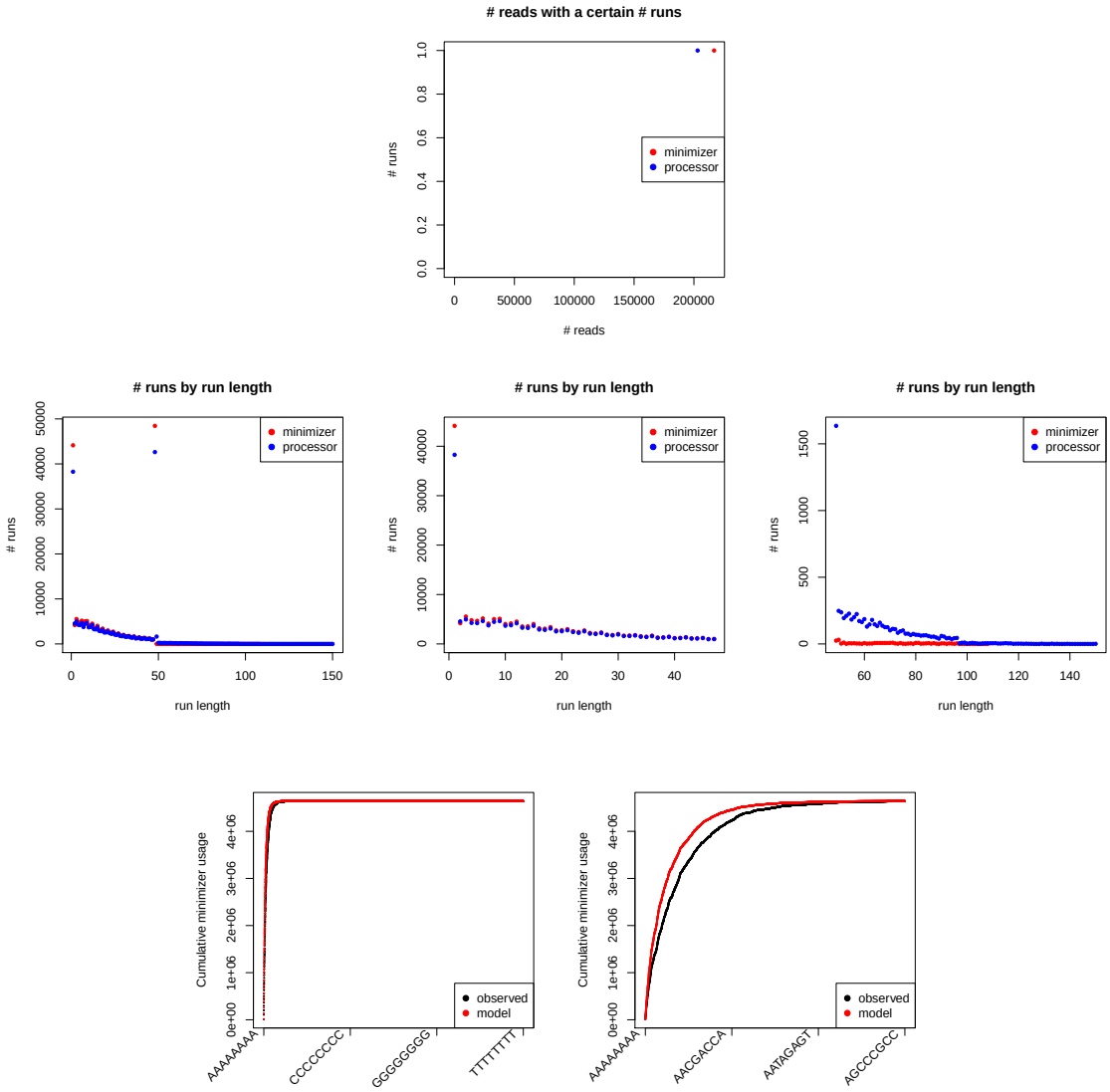




Train on (B), evaluate on (B), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 4639621
# mmers (with repeats) in evaluation data = 4639668

# distinct minimizers observed = 4619
Highest minimizer observed = AGCCCGCC

# runs of same minimizer = 216870
# runs of same processor = 203104

average minimizer run length = 21.393600 +/- 18.343300   median = 17.000000   mode = 48.000000
average processor run length = 22.843600 +/- 19.575200   median = 19.000000   mode = 48.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0       289980 6.250079  12743         22.7560 19.6202
1       289973 6.249929  12771         22.7056 19.4651
2       289976 6.249993  12918         22.4474 19.4810
3       289977 6.250015  12501         23.1963 19.6628
4       289973 6.249929  12820         22.6188 19.5225
5       289976 6.249993  12546         23.1130 19.7868
6       289985 6.250187  12663         22.9002 19.6254
7       289974 6.249950  12755         22.7341 19.6611
8       289974 6.249950  12665         22.8957 19.6035
9       289976 6.249993  12608         22.9994 19.5600
10      289974 6.249950  12721         22.7949 19.5296
11      289975 6.249972  12830         22.6013 19.5107
12      289979 6.250058  12604         23.0069 19.4791
13      289978 6.250036  12672         22.8834 19.6806
14      289976 6.249993  12447         23.2969 19.4543
15      289975 6.249972  12840         22.5837 19.5515

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length   std dev
AAAAAAAA       10770    0.2321310    224         48.0804  0.272454
AAAAAAAC       13499    0.5230815    348         38.7902 18.811800
AAAAAAAG       11529    0.7715716    314         36.7166 20.858400
AAAAAAAT       20102    1.2048398    497         40.4467 17.559400
AAAAAACA       14837    1.5246288    408         36.3652 20.345600
AAAAAACC       11959    1.7823870    358         33.4050 21.868900
AAAAAACG       13887    2.0817002    379         36.6412 20.067200
AAAAAACT       11413    2.3276901    281         40.6157 17.017900
AAAAAAGA       15503    2.6618338    421         36.8242 19.714400
AAAAAAGC       20233    3.0979255    546         37.0568 19.916000

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACTGAGCA           3     99.99950      1               3       0
ACTGAGGA           1     99.99953      1               1       0
ACTGATGC           3     99.99959      1               3       0
ACTGCCCG           4     99.99968      1               4       0
```

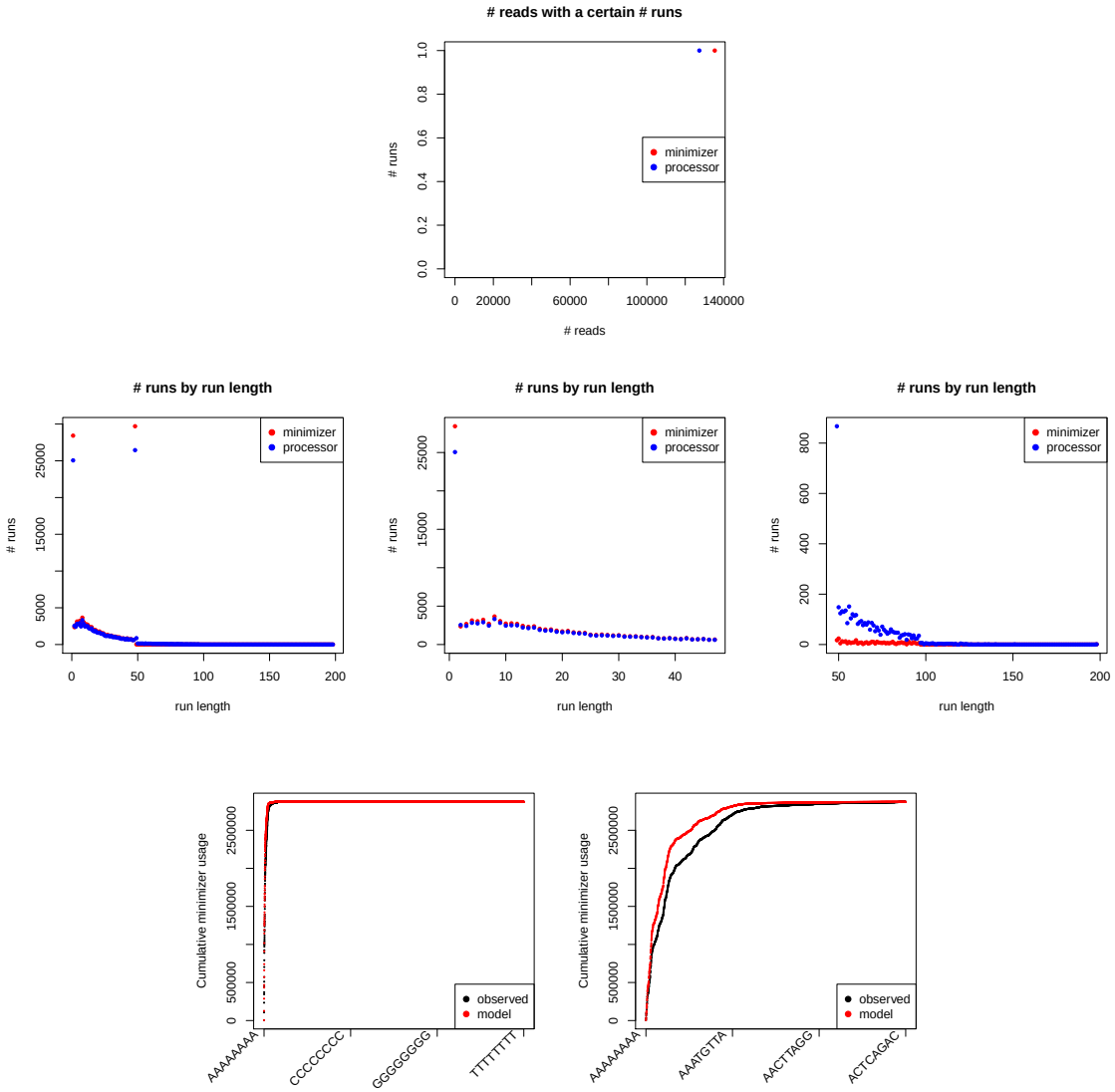
```
ACTGCGCG          2     99.99972      1              2       0
ACTGCGGC          1     99.99974      1              1       0
ACTGGCGC          2     99.99978      1              2       0
ACTTCATC          4     99.99987      1              4       0
AGATCCGC          3     99.99994      1              3       0
AGCCCGCC          3    100.00000      1              3       0

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
             % kmers     % mmers
A------- 100.0000000 43.08198345
AA------  99.7002988 14.04854830
AAA-----  81.0667509  4.67436463
AAAA----  39.4785695  1.52181147
AAAAA---  15.9173562  0.49846239
AAAAAA--   5.0087281  0.13798401
AAAAAAA-   1.2048398  0.03045477
AAAAAAAA   0.2321310  0.00521589
```



Train on (B), evaluate on (C), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 2872861
# mmers (with repeats) in evaluation data = 2872908

# distinct minimizers observed = 2858
Highest minimizer observed = ACTCAGAC

# runs of same minimizer = 135377
# runs of same processor = 127344

average minimizer run length = 21.221200 +/- 18.423200   median = 17.000000   mode = 48.000000
average processor run length = 22.559800 +/- 19.609200   median = 18.000000   mode = 48.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0       174350 6.068863   8092         21.5460 19.2817
1       195692 6.811746   8289         23.6086 19.9457
2       203440 7.081443   9190         22.1371 19.6502
3       194251 6.761587   7961         24.4003 20.0725
4       193349 6.730190   8716         22.1832 19.8898
5       125665 4.374211   6024         20.8607 18.3914
6       188168 6.549847   8330         22.5892 19.3799
7       162854 5.668704   7699         21.1526 18.8568
8       156163 5.435801   6864         22.7510 19.7330
9       186908 6.505988   7994         23.3810 20.1007
10      183198 6.376849   8250         22.2058 18.8782
11      182698 6.359444   7775         23.4981 20.0085
12      196988 6.856858   8416         23.4064 20.1584
13      190166 6.619394   8593         22.1303 19.5881
14      170706 5.942021   7673         22.2476 19.4835
15      168265 5.857053   7478         22.5013 19.5948

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length   std dev
AAAAAAAA        4566    0.1589356     95         48.0632  0.244537
AAAAAAAC       13696    0.6356729    310         44.1806 13.032200
AAAAAAAG       17216    1.2349362    389         44.2571 13.051100
AAAAAAAT       31666    2.3371823    699         45.3019 12.565100
AAAAAACA       18748    2.9897722    536         34.9776 21.101000
AAAAAACC        5450    3.1794786    141         38.6525 18.534900
AAAAAACG        6732    3.4138094    201         33.4925 21.152800
AAAAAACT        9651    3.7497463    283         34.1025 21.060000
AAAAAAGA       27126    4.6939619    752         36.0718 20.532700
AAAAAAGC       17624    5.3074270    443         39.7833 18.641200

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACACTGGT           6     99.99575      1               6 0.00000
ACAGACTG           1     99.99579      1               1 0.00000
ACAGATAC           9     99.99610      1               9 0.00000
ACAGCCCC          30     99.99715      5               6 0.00000
```

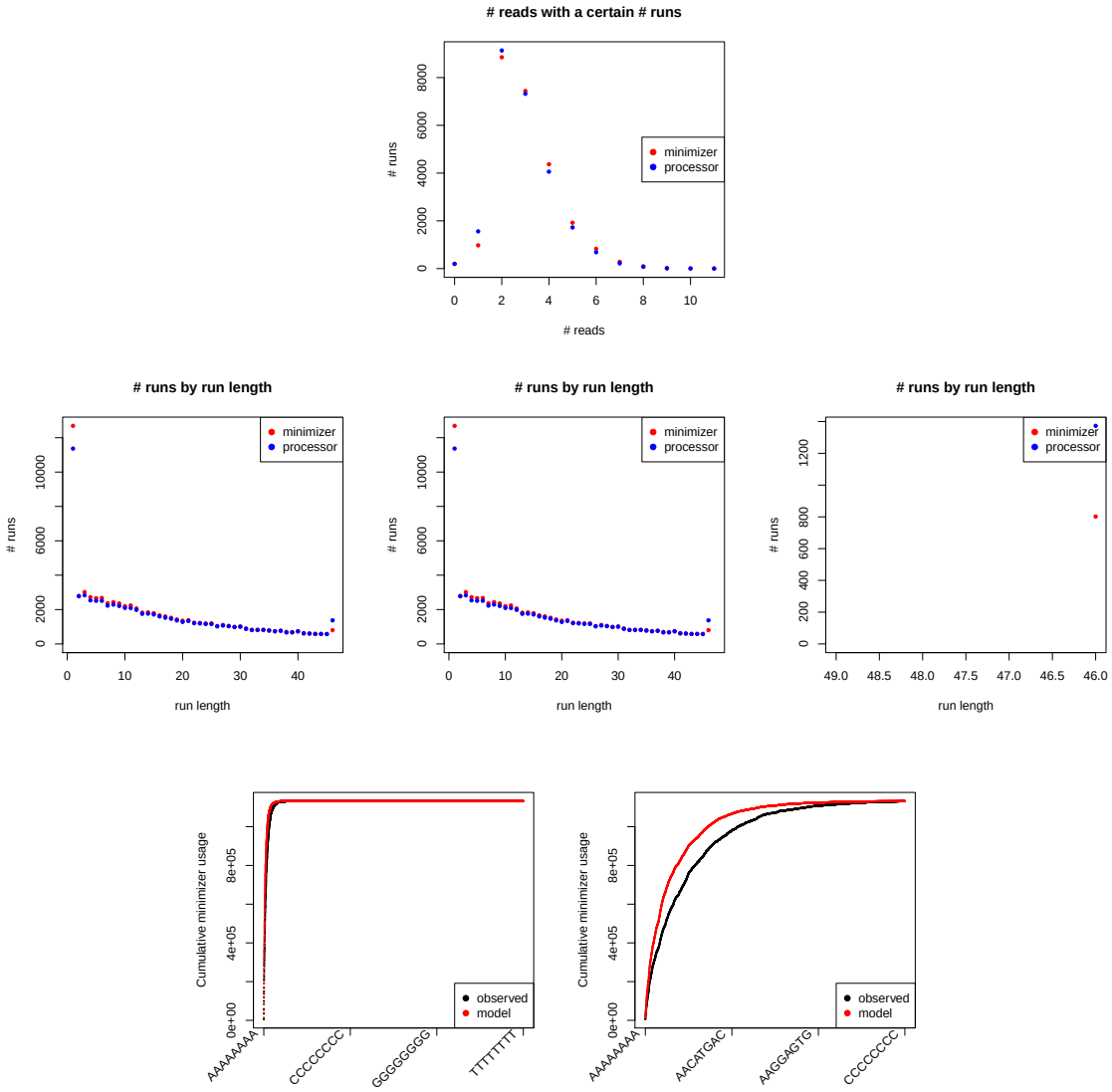
```
ACAGCGAC          36     99.99840      3             12 6.00000
ACAGGGGC           7     99.99864      1              7 0.00000
ACATAGGC           6     99.99885      1              6 0.00000
ACCAGCTG          14     99.99934      2              7 5.65685
ACGATATC           1     99.99937      1              1 0.00000
ACTCAGAC          18    100.00000      1             18 0.00000

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
             % kmers      % mmers
A------- 100.0000000 56.195534281
AA------  99.9954053 23.083335770
AAA-----  95.6739292  8.660701979
AAAA----  63.9027436  3.038976535
AAAAA---  29.9283188  1.005009558
AAAAAA--   9.9265158  0.270596900
AAAAAAA-   2.3371823  0.052908064
AAAAAAAA   0.1589356  0.003515602
```



Train on (C), evaluate on (A), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 1131857
# mmers (with repeats) in evaluation data = 2308330

# distinct minimizers observed = 3720
Highest minimizer observed = CCCCCCCC

# runs of same minimizer = 76127
# runs of same processor = 73104

average minimizer run length = 14.868000 +/- 12.744400   median = 12.000000   mode = 1.000000
average processor run length = 15.482800 +/- 13.087400   median = 13.000000   mode = 1.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0        66119 5.841639   4214         15.6903 13.1514
1        76442 6.753680   4907         15.5782 13.1916
2        69996 6.184173   4577         15.2930 13.0286
3        65865 5.819198   4334         15.1973 12.9311
4        68965 6.093084   4412         15.6312 13.1553
5        88258 7.797628   5472         16.1290 13.3012
6        74200 6.555598   4830         15.3623 12.9640
7        81634 7.212395   5113         15.9660 13.3371
8        63147 5.579062   4252         14.8511 12.8245
9        59208 5.231050   3983         14.8652 12.8974
10       64947 5.738092   4294         15.1251 12.9528
11       72810 6.432791   4643         15.6817 13.0376
12       56059 4.952834   3816         14.6905 12.7853
13       80194 7.085171   4945         16.2172 13.3844
14       72706 6.423603   4707         15.4464 13.0935
15       71307 6.300001   4605         15.4847 13.0558

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AAAAAAAA        5538    0.4892844    231         23.9740 14.0099
AAAAAAAC        3724    0.8183013    187         19.9144 14.5048
AAAAAAAG        3778    1.1520890    181         20.8729 14.2107
AAAAAAAT        6881    1.7600280    319         21.5705 14.5213
AAAAAACA        4641    2.1700621    240         19.3375 15.2266
AAAAAACC        4298    2.5497921    205         20.9659 14.5270
AAAAAACG        3484    2.8576048    155         22.4774 14.6887
AAAAAACT        2917    3.1153229    132         22.0985 14.4417
AAAAAAGA        3796    3.4507009    200         18.9800 15.3505
AAAAAAGC        4295    3.8301658    227         18.9207 14.6856

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
AGGGGGGG           5     99.96925      1             5.0 0.00000
CAAGCCGA           1     99.96934      1             1.0 0.00000
CACCCCCC           1     99.96943      1             1.0 0.00000
CCACCCCC           1     99.96952      1             1.0 0.00000
```

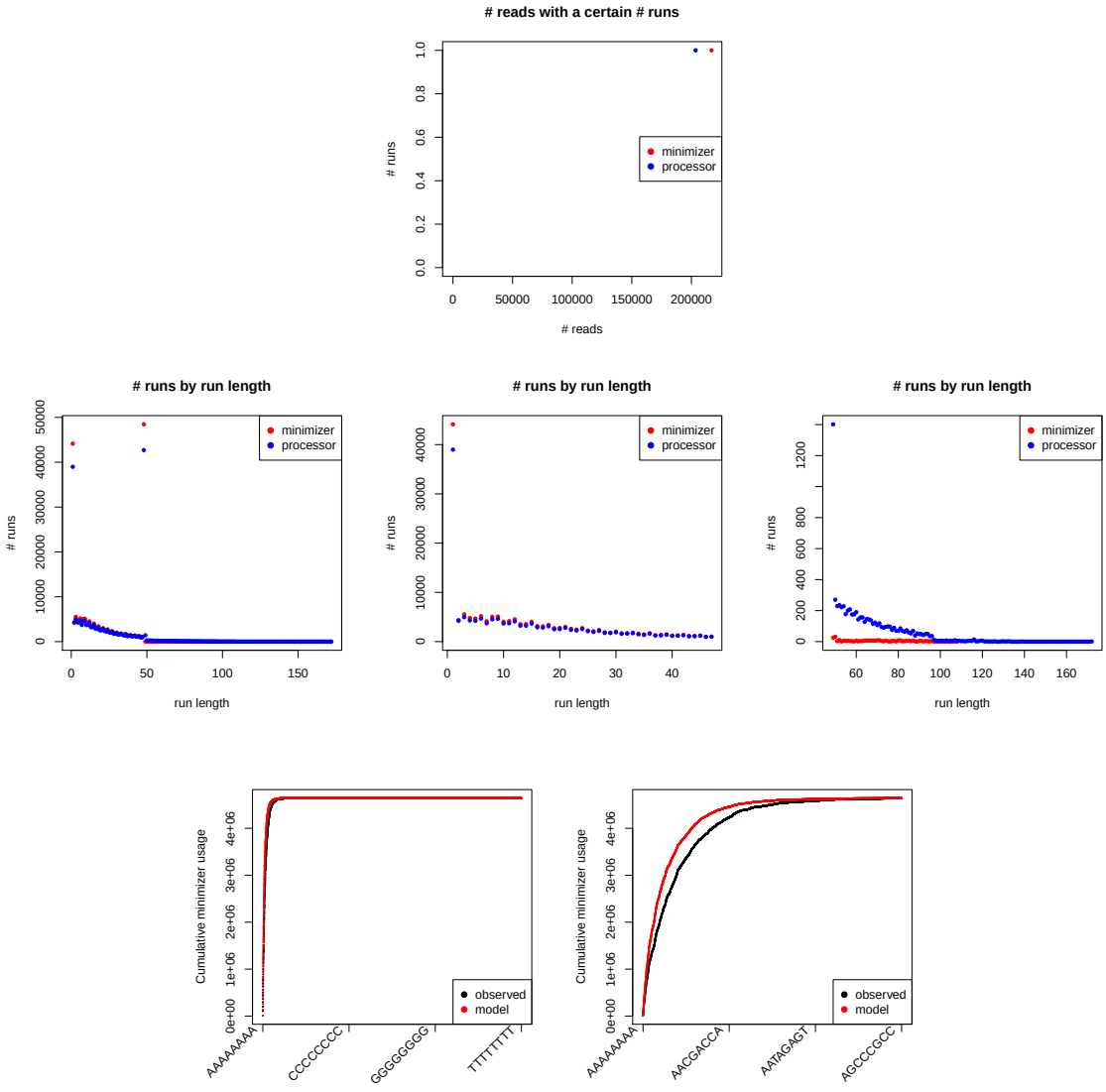
```
CCCACCCC           1     99.96961      1            1.0 0.00000
CCCCACCC           1     99.96970      1            1.0 0.00000
CCCCCACC           1     99.96978      1            1.0 0.00000
CCCCCCAC           1     99.96987      1            1.0 0.00000
CCCCCCCA           1     99.96996      1            1.0 0.00000
CCCCCCCC         340    100.00000      8           42.5 6.63325

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
           % kmers     % mmers
A------- 99.9692541 44.24228771
AA------ 99.6572005 14.57581888
AAA----- 81.6119881  4.91047641
AAAA---- 41.4364182  1.67458726
AAAAA--- 17.6879235  0.58891060
AAAAAA--  6.0208136  0.18931435
AAAAAAA-  1.7600280  0.06108312
AAAAAAAA  0.4892844  0.02473650
```



Train on (C), evaluate on (B), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 4639621
# mmers (with repeats) in evaluation data = 4639668

# distinct minimizers observed = 4619
Highest minimizer observed = AGCCCGCC

# runs of same minimizer = 216870
# runs of same processor = 203522

average minimizer run length = 21.393600 +/- 18.343300   median = 17.000000   mode = 48.000000
average processor run length = 22.796700 +/- 19.624600   median = 19.000000   mode = 48.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0       254099 5.476719  11378         22.3325 19.2410
1       320810 6.914573  13877         23.1181 19.9067
2       283977 6.120694  12852         22.0959 19.4424
3       255282 5.502217  11792         21.6487 19.1508
4       283444 6.109206  12235         23.1667 19.7837
5       359366 7.745590  14915         24.0943 20.0675
6       315068 6.790813  13573         23.2128 19.7304
7       333081 7.179056  13994         23.8017 20.0184
8       256988 5.538987  11849         21.6886 19.0300
9       249063 5.368176  11305         22.0312 19.2465
10      267655 5.768898  11953         22.3923 19.4376
11      309240 6.665200  13047         23.7020 19.8206
12      225836 4.867553  10427         21.6588 18.8173
13      321535 6.930200  14106         22.7942 19.9742
14      290303 6.257041  12867         22.5618 19.6066
15      313874 6.765078  13352         23.5076 19.9150

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length   std dev
AAAAAAAA       10770    0.2321310    224         48.0804  0.272454
AAAAAAAC       13499    0.5230815    348         38.7902 18.811800
AAAAAAAG       11529    0.7715716    314         36.7166 20.858400
AAAAAAAT       20102    1.2048398    497         40.4467 17.559400
AAAAAACA       14837    1.5246288    408         36.3652 20.345600
AAAAAACC       11959    1.7823870    358         33.4050 21.868900
AAAAAACG       13887    2.0817002    379         36.6412 20.067200
AAAAAACT       11413    2.3276901    281         40.6157 17.017900
AAAAAAGA       15503    2.6618338    421         36.8242 19.714400
AAAAAAGC       20233    3.0979255    546         37.0568 19.916000

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACTGAGCA           3     99.99950      1               3       0
ACTGAGGA           1     99.99953      1               1       0
ACTGATGC           3     99.99959      1               3       0
ACTGCCCG           4     99.99968      1               4       0
```

```
ACTGCGCG          2     99.99972      1              2       0
ACTGCGGC          1     99.99974      1              1       0
ACTGGCGC          2     99.99978      1              2       0
ACTTCATC          4     99.99987      1              4       0
AGATCCGC          3     99.99994      1              3       0
AGCCCGCC          3    100.00000      1              3       0

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
            % kmers     % mmers
A------- 100.0000000 43.08198345
AA------  99.7002988 14.04854830
AAA-----  81.0667509  4.67436463
AAAA----  39.4785695  1.52181147
AAAAA---  15.9173562  0.49846239
AAAAAA--   5.0087281  0.13798401
AAAAAAA-   1.2048398  0.03045477
AAAAAAAA   0.2321310  0.00521589
```



Train on (C), evaluate on (C), minimum s = AAAAAAAA

```
k = 55
m = 8
# processors = 16

# distinct mmers = 65536
# distinct mmers modulo duality = 32896

# kmers (with repeats) in evaluation data = 2872861
# mmers (with repeats) in evaluation data = 2872908

# distinct minimizers observed = 2858
Highest minimizer observed = ACTCAGAC

# runs of same minimizer = 135377
# runs of same processor = 127468

average minimizer run length = 21.221200 +/- 18.423200   median = 17.000000   mode = 48.000000
average processor run length = 22.537900 +/- 19.573000   median = 18.000000   mode = 48.000000

Processor usage over kmers (with repeats) in evaluation data
Proc #
   # kmer uses        % # runs mean run length std dev
0       179634 6.252791   7892         22.7615 19.5492
1       179628 6.252582   8283         21.6863 19.4842
2       179432 6.245760   8037         22.3257 19.5036
3       179418 6.245273   7876         22.7803 19.4885
4       179419 6.245307   8085         22.1916 19.5969
5       179413 6.245099   7757         23.1292 19.8411
6       179544 6.249658   7917         22.6783 19.4577
7       179428 6.245621   7963         22.5327 19.6022
8       179735 6.256307   8049         22.3301 19.6769
9       180101 6.269047   7920         22.7400 19.8171
10      179428 6.245621   8139         22.0455 19.6107
11      179425 6.245516   7798         23.0091 19.5013
12      179415 6.245168   7952         22.5622 19.5168
13      179548 6.249798   8034         22.3485 19.4560
14      179878 6.261284   7932         22.6775 19.5751
15      179415 6.245168   7834         22.9021 19.4557

Minimizer usage over kmers (with repeats) in evaluation data
First 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length   std dev
AAAAAAAA        4566    0.1589356     95         48.0632  0.244537
AAAAAAAC       13696    0.6356729    310         44.1806 13.032200
AAAAAAAG       17216    1.2349362    389         44.2571 13.051100
AAAAAAAT       31666    2.3371823    699         45.3019 12.565100
AAAAAACA       18748    2.9897722    536         34.9776 21.101000
AAAAAACC        5450    3.1794786    141         38.6525 18.534900
AAAAAACG        6732    3.4138094    201         33.4925 21.152800
AAAAAACT        9651    3.7497463    283         34.1025 21.060000
AAAAAAGA       27126    4.6939619    752         36.0718 20.532700
AAAAAAGC       17624    5.3074270    443         39.7833 18.641200

Last 10 with nonzero usage:
Minimizer
         # kmer uses cumulative % # runs mean run length std dev
ACACTGGT           6     99.99575      1               6 0.00000
ACAGACTG           1     99.99579      1               1 0.00000
ACAGATAC           9     99.99610      1               9 0.00000
ACAGCCCC          30     99.99715      5               6 0.00000
```

```
ACAGCGAC          36     99.99840      3             12 6.00000
ACAGGGGC           7     99.99864      1              7 0.00000
ACATAGGC           6     99.99885      1              6 0.00000
ACCAGCTG          14     99.99934      2              7 5.65685
ACGATATC           1     99.99937      1              1 0.00000
ACTCAGAC          18    100.00000      1             18 0.00000

Permutation mask prefix percentiles
  % kmers with minimizers matching the prefix
  % canonical mmers matching the prefix
             % kmers      % mmers
A------- 100.0000000 56.195534281
AA------  99.9954053 23.083335770
AAA-----  95.6739292  8.660701979
AAAA----  63.9027436  3.038976535
AAAAA---  29.9283188  1.005009558
AAAAAA--   9.9265158  0.270596900
AAAAAAA-   2.3371823  0.052908064
AAAAAAAA   0.1589356  0.003515602
```