Freie Universität Berlin
Department of mathematics and computer science

Bachelor thesis

Reducing Interleaved Bloom Filter space usage by estimating k-mer multiset cardinalities

Felix Droop

Supervisors

Prof. Dr. Knut Reinert Svenja Mehringer Enrico Seiler

Reviewers

Prof. Dr. Knut Reinert Prof. Dr. László Kozma

Contents

0.1 Abstract

The Interleaved Bloom Filter (IBF) developed by the Reinert group is a data structure used in state-of-the-art bioinformatics tools for the analysis of sequencing data. It approximately classifies reads into parts (bins) of a sequence data base. For the IBF to be space efficient, these bins need to be of similar size.

A balancing algorithm for user given and possibly unbalanced bins is described. It is evaluated if and how this algorithm can be improved by using approximations of the bins' shared k-mer content.

Experiments show no improvements when all bins share relatively few k-mers. However, when clusters of highly similar bins exist, reductions of up to 60% are in a parameter that directly influences the IBF memory consumption are observed.

1 Introduction

Due to advances in the technology of next-generation short-read sequencing (NGS), the amount of available genomic data has grown rapidly in the last decades. Producing raw NGS data is continuously becoming cheaper, easier and less time consuming. From 1982 to the present, the number of bases in one of the most important public databases GenBank has doubled approximately every 18 months [1].

This development poses both a great opportunity and a challenge. The new abundance of data could allow scientists to further understand the effects of genetic variation. A paradigm shift is happening from viewing a 'reference genome' as just a single linear sequence to viewing it as a collection of multiple sequences that captures the genetic variation, also called a 'pan-genome' [2]. Such a reference genome is needed in most kinds of NGS analysis to approximately search reads from the raw data and gain information about their origin in the biological sequences of interest (*read mapping*).

Established bioinformatics tools struggle with the increasingly high requirements of this type of analysis, both conceptually and in terms of time and space efficiency. It is often desirable to use whole data bases consisting of several hundreds of gigabases as the reference pan-genome. Constructing a single search structure like an FM-index for this amount of data is not possible in reasonable time with today's hardware. To make matters even more difficult, many sequences are added to the databases every day, which adds the necessity to be able to update the index without recomputing it on every update operation.

Therefore, novel data structures and methods are needed. One of such is the Interleaved Bloom Filter (IBF) [3]. It can be used to approximately classify reads into parts (bins) of a sequence data base, as needed for metagenomic analysis. The tool ganon [4] applies the IBF to provide precise and ultra-fast short read classifications on an updatable reference pan-genome. In combination with a full-text index, the IBF serves as a query filter. In DREAM-Yara [5], the IBF was used in this way to enhance the read mapper Yara to handle data bases well beyond what was possible before. Both of these applications were shown to be competitive or superior in terms of accuracy, time and space efficiency compared to other state-of-the-art programs regarding construction, search and update queries [see 5, 4, 3, benchmarks]. Figure 1.1 shows an example workflow that involves the IBF data structure.

1.1 Focus Of This Work

An IBF is a collection of Bloom Filters (BFs) [6]. The IBF and BF will be introduced thoroughly in the prerequisites, section 1.2. It follows a brief summary of the drawback of the IBF that this work tries to improve on.

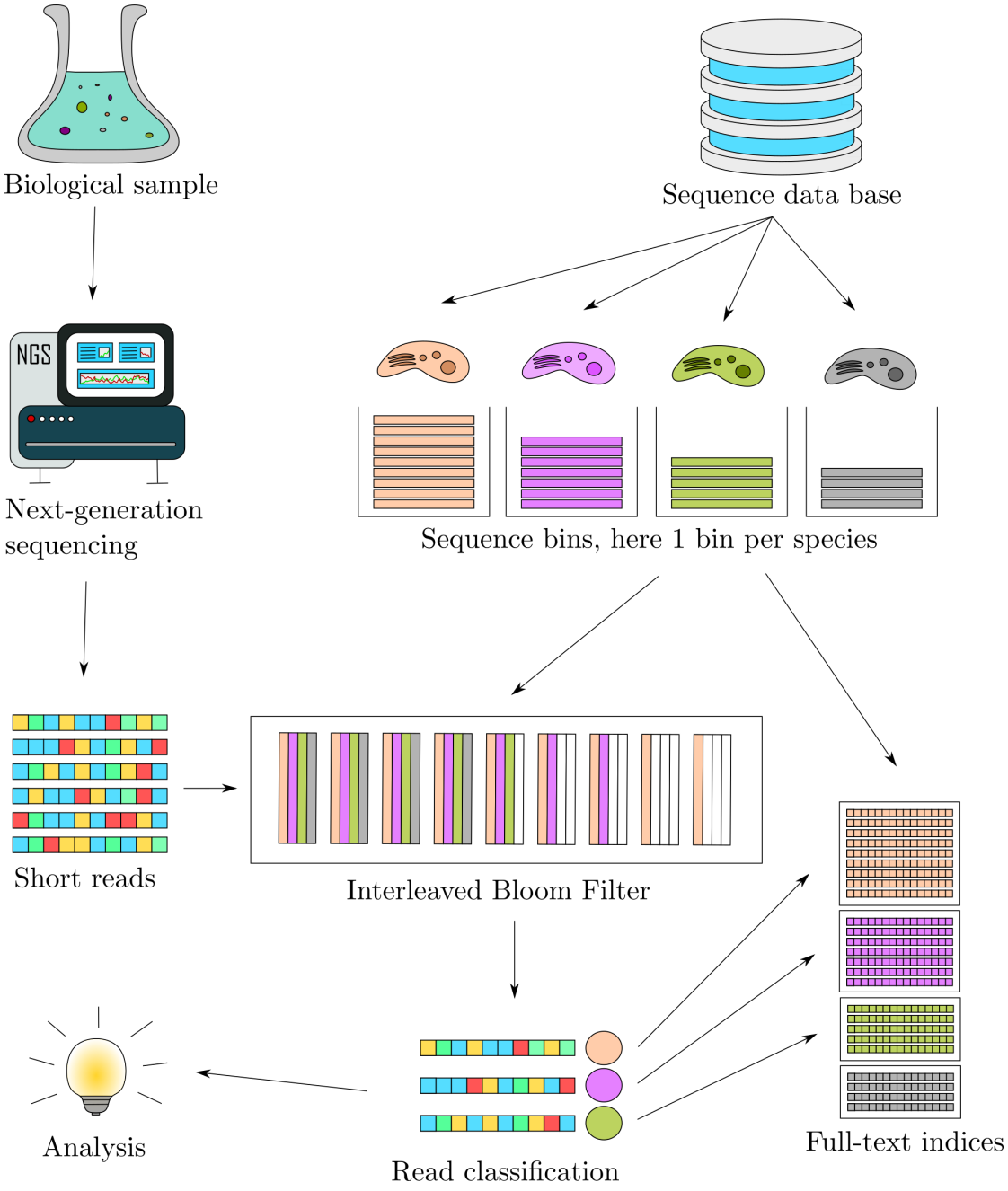


Figure 1.1: An example workflow that involves the IBF data structure. Short reads of DNA sequence are generated from a biological sample using NGS. The reads are then classified by the IBF into bins of sequences from a reference data base. The classifications are used for further analysis and read mapping via full-text indices.

In the IBF, every individual BF stores the k-mer set [1] of a bin of sequences. The BFs are not stored disjointedly in an array. They are instead *interleaved*, which leads to a better running time of search queries (for details see prerequisites). The drawback of the interleaving is that all the individual BFs have to be of equivalent size. This leads to a higher space usage, because the sizes of BFs of bins with smaller sequences must be scaled up to the size of the largest BF.

This drawback is negligible if the user-given sequence bins are already of approximately even size. Since this cannot be guaranteed for many applications, the Algorithmic Bioinformatics group of the Freie Universität Berlin around Professor Reinert has designed an algorithm that computes a second set of bins by splitting and merging the original user-given bins [7]. The goal of the algorithm is to minimize the maximum bin size in this second set, because the internal IBF is built on this set of bins. Further adjustments have to be made to allow returning read classifications regarding the original user-given bins. The result is a data structure that consists of multiple IBFs on multiple levels, the hierarchical IBF. The new algorithm and data structure are not yet published and will be introduced in detail in the Methods section.

This work aims to adjust the algorithm such that it takes into account the sequence similarity of the bins while merging original bins. Focusing the merging on similar sequences might reduce the maximum number of distinct k-mers in any bin (maximum bin size). Since the BF sizes are proportional to that value, a more balanced IBF with less space consumption will be the result.

1.2 Prerequisites

The following section introduces existing mathematical methods, algorithms and data structures that are used in the contributions of this work.

1.2.1 Multisets And Jaccard Index

A *multiset* is a collection of elements where each element may occur repeatedly. The *size* of a multiset is the number of elements contained in it, counting multiple instances of the same element individually.

The *underlying set* of a multiset is the set of distinct elements in the multiset. The *cardinality* of a multiset is the number of elements in its underlying set. The cardinality of a multiset is smaller than or equal to its size.

The Jaccard index J is a statistic that measures the similarity between two sets A and B:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{1.1}$$

[1] The k-mer set of a sequence consists of all continuous subsequences of length k.

The Jaccard distance d_J is the corresponding metric that measures the *dis*similarity between sets:

$$d_J(A,B) = 1 - J(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} = 2 - \frac{|A| + |B|}{|A \cup B|} \tag{1.2}$$

The Jaccard index and distance can be generalized for multisets. However, only the Jaccard distances of the underlying sets of multisets are used in this work. Such Jaccard distances can be computed directly or indirectly by computing the cardinalities of the individual multisets and the cardinality of their union (eq. 1.2).

Computing the cardinality of a multiset exactly can be done in time proportional to its size and with space consumption proportional to its cardinality. An implementation using a hash table achieves these characteristics and is suitable for input multisets with relatively small cardinalities. The space requirements become a problem for very large cardinalities. For example, if roughly half of the theoretically possible 2^{40} 20-mers are present in a large data base of DNA sequences, computing the exact number of distinct 20-mers requires roughly 2.5 TB of space without counting the overhead of any data structures used.

There exist approximation algorithms that allow estimating cardinalities of very large multisets using only constant amounts of memory, for example the HyperLogLog algorithm.

1.2.2 HyperLogLog

The HyperLogLog (HLL) algorithm [8] approximates the number of distinct elements in a data stream. In other words, it approximates the cardinality of multisets. The algorithm in its simplest form consists of building a small data structure called *sketch* that keeps track of a statistic on the input data. The sketch is built in a single pass over the data. Afterwards, the cardinality estimate can be retrieved or the sketch can be used in further computations that involve adding more elements or merging multiple sketches.

The first step of the algorithm is to convert the data stream into a stream of uniformly distributed fixed-size hash values by applying a suitable hash function. The length of the hash values q determines the largest cardinality that can be estimated. 64-bit hash values are sufficient for most practical applications including the application in this work.

The estimate is based upon the observation that every hash value in the stream has a probability of $1/2^p$ of containing p leading zeros. Observing such a value is a more or less likely indication that the stream contains at least 2^p distinct elements. One could compute the maximum number of leading zeros p_{max} of any hash value in the stream and estimate the cardinality by $2^{p_{max}}$.

This estimation method alone has a far too high variance to be useful. For example, a single unlucky hash value of all zeros would instantly produce the highest possible cardinality estimate 2^q. The variance is reduced by splitting the data stream into $m = 2^b$ substreams and keeping track of p_{max} for every substream, where $b \in [4, q-1]$. The

splitting is achieved by using the first b bits of the hash value as the index for the corresponding substream in the HLL sketch.

The sketch consists of m registers that each hold the current p_{max} for a substream. These registers must be able to store the largest possible p_{max}, which is $q - b$. Therefore their size has to be at least $\log_2(q - b)$ bits. Ignoring b, the register size is $\log_2 \log_2$ of the maximum cardinality 2^q that can be estimated, hence the name of the algorithm. In many implementations of the HLL algorithm, bytes are used as registers for simplicity.

To retrieve the cardinality estimate for the whole data stream, the values from the registers are combined using the *harmonic mean* and bias correction. This takes time in $O(m)$. For more details, please refer to the original HLL paper [8].

Contrary to exact methods that solve the cardinality estimation problem, the size of HLL sketches in memory is not proportional to the number of distinct elements in the stream. The number of registers m can instead be chosen to fit the needs of a given application. Larger sketches provide more accurate estimates. The relative accuracy of the estimate is typically about $1.04/\sqrt{m}$. For example, a sketch of size 1.5 KB should be able to estimate cardinalities way beyond 10^9 with a typical accuracy of 2% [8].

To verify this, the HyperLogLog implementation used in the later examined program was tested with randomly generated DNA sequences (see Figure 1.2). This implementation uses bytes as registers, instead of the only necessary 6 bits. It therefore is slightly less space efficient than theoretically possible. The results show that the majority of estimates indeed have relative errors as expected.

Finally, HLL sketches can be merged in time $O(m)$. This is done by computing the pairwise maximum between all registers of the two sketches. The resulting sketch is equivalent to a single sketch built on the two streams upon which the predecessor sketches were built. The ability to efficiently merge HLL sketches is extremely important for this work.

All operations of the HLL can be implemented using only few low level operations and are very efficient. Exact methods for computing multiset cardinalities are generally much slower, even if the asymptotic bounds are equivalent.

1.2.3 Bloom Filters

A Bloom Filter (BF) [6] is an approximate set membership data structure. It consists of a bit array of fixed size m and a set of h hash functions. These hash functions map elements of some domain to the set $\{0, \ldots, m-1\}$. All bits of the bit array are initially set to 0.

An element is *inserted* into the BF by computing the hashes of it with all of the hash functions and setting the respective bits of the bit array to 1.

To check whether a BF *contains* an element, again all hashes are computed. If all of the respective bits of the bit array are set to 1, the BF is assumed to contain the element. This is an approximation, because false positives (FPs) are possible. The bits might have been set to 1 by the insertion of other values. On the contrary, no false negatives are possible.

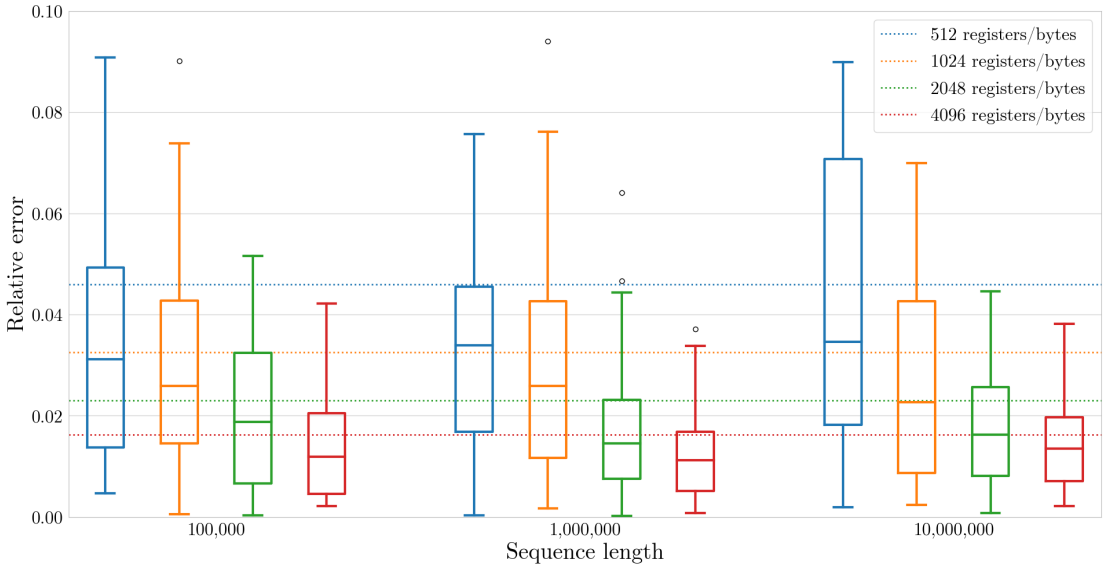


Figure 1.2: Every boxplot consists of relative errors of HyperLogLog estimates of the number of 20-mers in 30 randomly generated DNA sequences. The sizes of the sequences are given on the x-axis. The dotted horizontal lines indicate the expected relative error due to the formula $1.04/\sqrt{m}$ [8], where m is the number of registers with a size of one byte each. The colors of boxplots and lines allow to distinguish between the number of HyperLogLog registers in the respective experiments.

The probability for a FP in a *contains* query ε is often given as the following, where n is the number of elements inserted into the BF:

$$\varepsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{hn}\right)^h \approx \left(1 - e^{-hn/m}\right)^h \tag{1.3}$$

This formula is an approximation, because it assumes that the probabilities of each bit in the BF being set to 1 are independent.

The number of hash functions h must be a positive integer. Ignoring this constraint, the FP probability is minimized by the following value for h:

$$h = \frac{m}{n} \ln 2 \tag{1.4}$$

Inserting this value back into the FP probability formula (eq. 1.3) and simplifying gives an approximation for the number of bits m in terms of the number of inserted elements n and the FP probability ε:

$$\varepsilon = \left(1 - e^{-\left(\frac{m}{n} \ln 2\right) n/m}\right)^{\frac{m}{n} \ln 2} \tag{1.5}$$

$$\iff m = -\frac{n \ln \varepsilon}{(\ln 2)^2} \tag{1.6}$$

The BF size m is proportional to the number of stored elements n, because for all FP probabilities ε between 0 and 1, $\ln \varepsilon$ is smaller than 0. It follows that for any expected n, m can be adjusted proportionally to achieve the desired ε.

Bloom filters are widely used because of the small query time and space efficiency. Since no false negatives are possible, they often serve as prefilters of queries when an exact look-up is expensive.

1.2.4 Interleaved Bloom Filter

An Interleaved Bloom Filter (IBF) [3] is a collection of Bloom Filters. Each of these BFs stores a different set of elements, all stemming from a common domain. These sets of elements are referred to as *bins* in the following.

The *contains* query of an IBF returns a bit array, which specifies whether an element is present in each of the internal BFs. The goal of the IBF data structure is to make this query efficient while consuming as little memory as possible.

The bit arrays of the individual BFs are not stored disjointedly after one another, but instead are *interleaved* (see Figure 1.3). The corresponding bits on the same indices are stored next to each other in a single large bit array. Alternatively, the IBF can be viewed as a large BF, where the bits are replaced by small bit arrays. These bit arrays are predecessors of the ones retrieved in the *contains* queries.

This way, only a single set of h hash functions is needed for the whole data structure. A *contains* query works as follows. The hash values are computed. The bit arrays of size b at those positions are retrieved and iteratively combined with bit-wise logical ANDs. The running time T of this query can be described as:

$$T(h, b, w) = h \cdot (c_h + c_{MOV} \cdot \lceil b/w \rceil) + (h - 1) \cdot (c_{AND} \cdot \lceil b/w \rceil) \in O\left(\frac{h \cdot b}{w}\right) \tag{1.7}$$

Where w is the word size. c_h, c_{MOV} and c_{AND} are constants that describe the running times of computing a hash value, locating a word in memory and copying it to a destination, and computing the bit-wise logical AND between two words, respectively. The $\lceil b/w \rceil$ in the formula is due to the fact that only full words can be handled by the machine.

The most important implications of equation 1.7 are that the running time of the *contains* query grows linearly in b and that it is most efficient when b is a multiple of the word size w. In practice, the running time is mostly influenced by the number of bins b and the computation of the hash functions.

An implementation with disjointed BFs would be much slower for two reasons. Firstly, hashes would have to be computed for every BF individually and not just for the single set of hash functions, if the BFs cannot use the same hash function because they differ in size. Secondly, it is not possible to to operate on single bits with maximum efficiency on today's common hardware. Handling complete bit arrays therefore saves a lot of running time. Cache effects and SIMD operations amplify this behavior.

The drawback of the interleaving is the additional space usage. To prevent the necessity for additional bookkeeping that would negate all running time benefits, the individ-

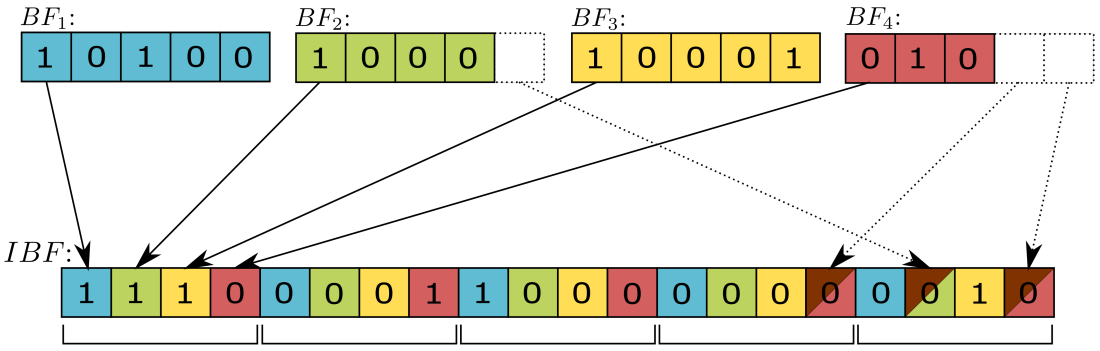


Figure 1.3: A visualization of how four Bloom Filters are turned into an Interleaved Bloom Filter. The four BFs store different numbers of elements and therefore have different sizes for a fixed FP rate. For each leftmost bit of the BFs the corresponding bit in the IBF is indicated by a continuous arrow. The overhead bits of BF_2 and BF_4 are shown with dotted lines and indicated by a brown triangle in the IBF. The IBF has a size of $5 \cdot 4 = 20$ while the BFs have a cumulative size of $5 + 4 + 5 + 3 = 17$.

ual BFs have to be of equivalent size. BFs storing smaller sets of elements have to be scaled up to the size of the largest one (see Figure 1.3). Scaling the larger ones down would inflate the FP rate. When all BFs are of equivalent size, simple random access is possible in the resulting interleaved bit array.

The IBF was designed and is mainly used in the field of bioinformatics to enable the analysis of large amounts of next generation sequencing data. To do so, the reference pan-genome is split into sequence bins. The exact way of splitting depends on the application. The k-mers of the sequences in the bins are then inserted into the IBF. After this building step, the data structure is used in conjunction with the k-mer counting lemma or a variant thereof to classify reads into the bins with error tolerance [see 3, Methods].

In some applications it is sufficient to get the classification for the reads [4]. For example, in metagenomic analysis one might only want to verify presence or absence of organisms on certain taxonomic levels to gain information about the original biological sample. Here, the IBF is the only data structure needed and its bins only have to represent the k-mer sets of the taxa in the reference pan-genome.

Other applications might want to perform sequence assembly. The IBF alone does not fulfill the requirements of such analysis, but it can still be of great use in combination with a full text index like an FM index [5]. It allows partitioning of the reference pan-genome. The bins then represent k-mer sets and continuous sequences upon which the indices are built. The IBF serves as query filter and distributor. This way, no index has to be built on the whole pan-genome. Such multiple smaller indices are much easier to

handle both in search and update queries.

In all applications, the bins can consist of k-mers from one up to many large sequences like genomes and be very uneven in size. Due to the sheer amount of data, it is infeasible to simply increase the sizes of internal BFs for smaller bins to match the size of the largest ones. Blindly redistributing the sequences is also not an option, because the exact composition of the bins might have importance for the conducted analysis. A solution for this problem is offered in the Methods section.

2 Methods

The ideas presented in the following have been developed by the Algorithmic Bioinformatics group of the Freie Universität Berlin around Professor Reinert [7] and have not yet been published. The contribution of this work is to introduce these concepts in the subsections 2.1 and 2.2 and evaluate a possible improvement for them in the other subsections of the Methods section, as well as in the Results and Discussion sections.

2.1 Algorithm Background

It is the goal to build an IBF for y user-given bins of sequences $C_0, \ldots, C_{y-1}$ with minimal space overhead. In the following, all sequence bins are conceptually viewed as k-mer multisets. Let $c_0, \ldots, c_{y-1}$ denote the number of unique k-mers (cardinality) in the respective bin. The user-given bins are called *user bins* in the following. An IBF built directly on this data would have a size in memory proportional to S_{user}, where:

$$S_{user} = \left(\max_{0 \leq i < y} c_i \right) \cdot y \tag{2.1}$$

To make this size small and the data structure efficient in terms of space usage, the user bins have to be of similar cardinality. Since this rarely is the case in practice, *technical bins* $B_0, \ldots, B_{x-1}$ are introduced. Let $b_0, \ldots, b_{x-1}$ denote the cardinality of the respective technical bin. The number of these bins x can either be user-provided or determined by other algorithms in tools that apply this algorithm.

A user bin can be split into any number of technical bins and any number of user bins can be merged into a single technical bin (see Figure 2.2 A). Mapping a single user bin to a single technical bin is considered splitting in the following. Both the splitting and merging are realized on the underlying k-mer sets of the originating user bins. The IBF will be built on the technical bins instead of the user bins. It has a size proportional to S_{tech}, where:

$$S_{tech} = \left(\max_{0 \leq i < x} b_i \right) \cdot x \tag{2.2}$$

The target of the following algorithm is to minimize S_{tech} by creating technical bins of similar cardinality and especially achieve $S_{tech} < S_{user}$. There are two important constraints to keep in mind:

1. A query result on the IBF built on technical bins must be efficiently transformable into a query result regarding the user bins. Otherwise the data structure could not function the way a user would expect.

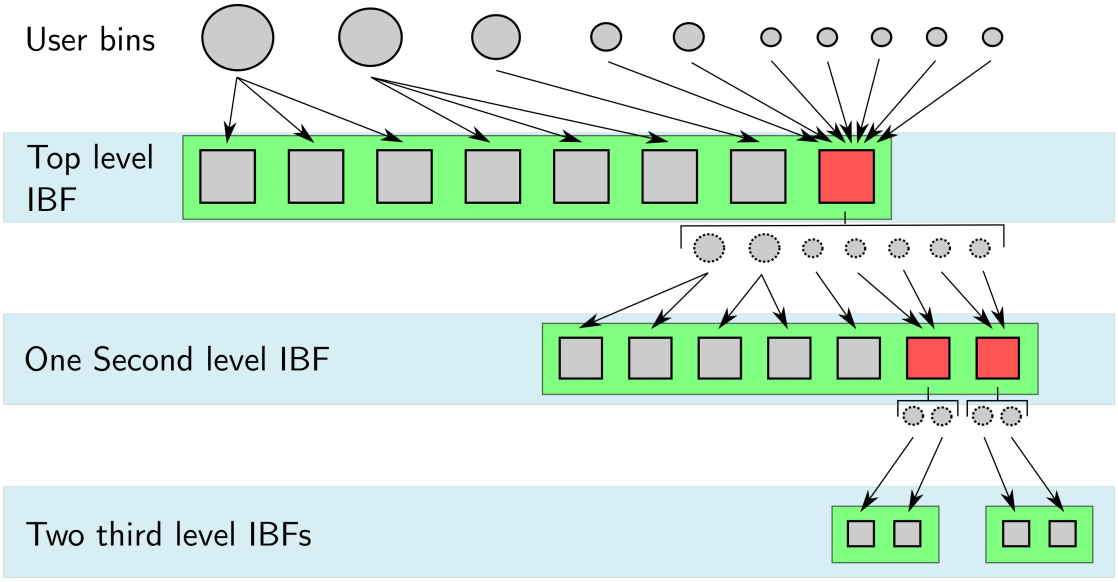


Figure 2.1: An example of a hierarchical IBF. The user bins are shown as grey circles. The individual IBFs are shown in green. Split bins are shown as grey squares and merged bins are shown red squares. For every merged bin, the user bins contained in it are shown below the bin with a dotted border.

2. x cannot be arbitrarily large, because the *contains* query running time of the IBF is in $O(\#bins)$, here $O(x)$ (eq. 1.7).

The first constraint can be easily resolved for technical bins created by splitting. One simply has to keep track of the user bin out of which the technical bins emerged. When a k-mer is classified into such a technical bin, it must also be present in the original user bin. More adjustments have to be made when a k-mer falls into a technical bin that was created by merging. One now has to classify the k-mer regarding the user bins contained in the merged bin. For this problem, an IBF can be used. For this IBF, the same procedure of splitting and merging can be recursively applied to minimize its S_{tech}.

The result of this recursive procedure is a data structure that consists of multiple IBFs on multiple levels, called a hierarchical IBF. On the highest level (in the following often referred to as *top level*), there is a single IBF whose technical bins contain all user bins between them. For every merged bin in the top level IBF, there is an IBF on the second level whose technical bins contain all user bins of this merged bin between them. For every merged bin in a second level IBF, there is an IBF on the third level, and so on. The IBFs on the lowest level do not contain merged bins. It holds that for every user bin, there is at least one technical bin that contains exclusively k-mers from this user bin on some level in the hierarchical IBF. An example visualization of a hierarchical IBF can be observed in Figure 2.1.

A *contains* query of the hierarchical IBF works as follows. First, a *contains* query is performed on the top level IBF. Then, for every merged bin the k-mer is classified into, a

contains query is performed recursively on the respective lower level IBF. The recursion keeps going until the k-mer is not classified into any merged bin. Finally, the k-mer is classified into the originating user bins of all split bins it got classified into during the recursive step-down.

To prevent negative impacts on the running time due to the cascading IBF queries, the recursion must be limited to only a few levels. How to choose this number of levels is not in the scope of this work. The focus lies on the algorithm which performs one recursive step. As a base case for the recurrence, only splitting is allowed on the lowest level.

It is important to allow merging on the other levels, because of the second constraint stated above. With a limited number of technical bins x and only splitting allowed, it may not be possible to achieve technical bins with relatively similar cardinalities, especially if many small user bins exist.

x can be chosen differently for the individual IBFs of the hierarchical IBF. One way of doing so is setting x to the minimum of a user-provided constant, e.g. $x_{max} = 1024$, and the number of user bins contained in the respective IBF. The details about the different ways of adjusting the number of technical bins in the hierarchical IBF are out of scope of this work.

The following section describes an algorithm that heuristically finds a good transformation from user bins to technical bins, i.e. one recursive step. The additional space usage on the lower levels is weighed against the benefit of merging on the upper level. An algorithm to perform the transformation on the lowest level with only splitting exists. It is omitted here, because it is a simpler version of the shown algorithm and the later discussed improvements focus on the merging.

Lastly, there is an important detail regarding the splitting of sequence bins. When the IBF is used as a stand-alone classifying data structure, the underlying k-mer sets of the bins can be partitioned into exactly even sized subsets. When the IBF is used together with a full text index, it is not that easy. The bins have to represent both k-mer sets and continuous sequences for the exact search after the IBF query. Therefore, it might not be possible the achieve a perfectly even splitting of the bins.

2.2 Hierarchical Binning Algorithm

The following algorithm is not described in the most general sense, but instead with the application as preprocessing for the IBF in mind.

The inputs of the algorithm are the desired number of technical bins x, the cardinalities $c_0, \ldots, c_{y-1}$ of the user bins and a real number $\alpha \geq 1$. α is a parameter which influences how much merging the algorithm does.

The output of the algorithm is a transformation from user bins $C_0, \ldots, C_{y-1}$ to technical bins $B_0, \ldots, B_{x-1}$. It heuristically minimizes S_{tech} of the upper level IBF plus an estimate of the S_{tech} of the lower level IBFs. One could use this result and recurse into each merged technical bin to obtain a blueprint for a full hierarchical IBF.

2.2.1 Outline And Definitions

The algorithm starts by sorting the user bins by cardinality in decreasing order. In the following, $c_0, \ldots, c_{y-1}$ are assumed to be sorted. The algorithm then proceeds to compute a mapping from user to technical bins with dynamic programming that reduces the overall space consumption. There are two simplifications:

1. Only merges of adjacent intervals in the fixed sorted order are allowed. It is expected that mainly small bins are merged. These are close to each other in the sorting. Considering all theoretically possible merges is not computationally feasible, because there are exponentially many subsets of the set of user bins.

2. The algorithm needs some way of estimating how merges of user bins impact lower recursive levels. Even with respect to the previous simplification, computing the S_{tech} values of lower level IBFs exactly is too computationally expensive. One would have to precompute the values for all quadratically many intervals of the user bin sequence. An efficient way of doing so is to compute the values for smaller intervals first. They can then be used in the computation of the values for the larger intervals. Still, this algorithm has a total running time in $O(y^2 \cdot T(x, y))$, where $T(x, y) \in \Theta(x \cdot y \cdot \max(x, y))$ is the running time of the hierarchical binning algorithm. An algorithm with this running time would be too slow to be computed in reasonable time on today's hardware with practically relevant input sizes.

   The S_{tech} values of lower level IBFs that arise through merging are instead estimated by the user-provided parameter α times the sum of cardinalities of user bins that are contained in the respective lower level IBF. α is a measure of the unevenness of technical bin cardinalities that directly influences the IBF overhead. For $\alpha = 1$ the estimate would be exact for a given IBF, if all technical bins have the same size and the IBF does not contain merged bins. Then, there is no IBF overhead and the IBF has the same size as a collection of normal BFs with the same number of hash functions and the same FP rate. When unevenness of technical bin cardinalities and merged bins are expected, α has to be larger than 1 for the estimate to be realistic.

For the dynamic programming scheme, the algorithm uses three $x \times y$ matrices:

- $M_{i,j}$ is the maximum technical bin cardinality on the current level, when the first $j + 1$ user bins are transformed into $i + 1$ technical bins.

- $L_{i,j}$ is the sum of cardinalities from merged user bins used to achieve the value in $M_{i,j}$. The values in this matrix are used to estimate the additional space usage of lower level IBFs.

- The third matrix is needed for trace-back. The details are omitted here, because it is a fairly standard dynamic programming trace-back and not relevant for the later proposed improvements.

2.2.2 Initialization

For $j = 0$, there is only a single user bin C_0 that can be split into all available technical bins. Since no merging is done, no lower level IBFs are needed.

$$\forall i \in \{0, \dots, x-1\} : M_{i,0} = \frac{c_0}{i+1},\ L_{i,0} = 0 \tag{2.3}$$

For $i = 0$ and $j \neq 0$, all user bins have to be merged into a single technical bin. The cardinality of the resulting merged bin is estimated by the sum of the cardinalities of the originating user bins. This is an upper bound, because the size of the union of sets is smaller than the sum of the individual sizes if the sets are not distinct.

The resulting additional space usage on lower levels is estimated by α times the sum of user bin cardinalities.

$$\forall j \in \{1, \dots, y-1\} : M_{0,j} = \sum_{g=0}^{j} c_g,\ L_{0,j} = \alpha \cdot M_{0,j} \tag{2.4}$$

2.2.3 Recurrence

For the cell $M_{i,j}$, the new user bin to process is C_j. The main recurrence consists of two cases (see Figure 2.2 B). In the first case, the technical bins $B_0, \dots, B_{i'}$ are used to store all the previous user bins $C_0, \dots, C_{j-1}$. The remaining user bin C_j is split into the technical bins $B_{i'+1}, \dots, B_i$:

$$v_{i,j} = \min_{i' \in \{0,\dots,i-1\}} \left(\underbrace{\max\left(M_{i',j-1}, \frac{c_j}{i-i'}\right)}_{\text{(I)}} \cdot (i+1) + \alpha \cdot \underbrace{L_{i',j-1}}_{\text{(II)}} \right) \tag{2.5}$$

To find the best intermediate result, the minimum over all possible values of i' is computed. Inside the large braces of the minimum is an estimate for the total size of the hierarchical IBF. The left summand is the S_{tech} of the current level IBF. It has $i+1$ technical bins. The term (I) denotes the cardinality of the largest technical bin. It is the maximum of the intermediate result for all the previous user bins and the cardinalities of the new split bins. This formula assumes that a perfectly even splitting of C_j is possible. The right summand is the estimate for the space usage of lower level IBFs with α. No new merged bins were added, therefore only the previous result (II) is needed.

In the second case, the technical bins $B_0, \dots, B_{i-1}$ are used to store the user bins

$C_0, \ldots, C_{j'}$. The remaining user bins $C_{j'+1} \ldots, C_j$ are merged into the technical bin B_i :

$$h_{i,j} = \min_{j' \in \{0,\ldots j-2\}^*} \left(\underbrace{\max\left(M_{i-1,j'}, \sum_{g=j'+1}^{j} c_g \right)}_{\text{(III)}} \cdot (i+1) + \alpha \cdot \underbrace{\left(L_{i-1,j'} + \sum_{g=j'+1}^{j} c_g \right)}_{\text{(IV)}} \right) \tag{2.6}$$

$$^*\text{only consider } j' \text{ if the user bin } C_{j'} \text{ was not truly split in the intermediate result in the cell } M_{i-1,j'}. \tag{2.7}$$

This time, the minimum over all possible values of j' is computed and a similar formula is minimized. The cardinality of the largest bin of the current level IBF is denoted by the term (III). It is the maximum of the previous intermediate result and the cardinality of the newly created merged bin. The latter is upper bounded by the sum of the originating user bins' cardinalities. The same sum is added to the estimate of the space usage of lower level IBFs in (IV).

The range for j' has an upper limit of $j-2$, because mapping a single user bin to a technical bin is already covered by the formula which covers the splitting (eq. 2.5). The constraint for j' (eq. 2.7) can save a lot of running time in practice and does not alter the result. The excluded values cannot produce the optimal value. If it was the optimum to split the user bin $C_{j'}$ into at least two technical bins in the intermediate result for the cell $M_{i-1,j'}$, then it will never be better to merge $C_{j'}$ into the newly available user bin together with the bins $C_{j'+1}, \ldots, C_j$ for the cell $M_{i,j}$. Otherwise it would have been more advantageous to map $C_{j'}$ to only a single technical bin in the intermediate result of $M_{i-1,j'}$. This works only because of the decreasing ordering of the user bins. In practice, the information about whether $C_{j'}$ was split into at least two technical bins (*truly* split) in the intermediate result for $M_{i-1,j'}$ can be extracted with a single look-up in the trace-back matrix.

Let m_v, l_v, m_h and l_h be the values of the terms (I), (II), (III), and (IV) for which the minima in $v_{i,j}$ and $h_{i,j}$ are achieved, respectively. The values of the cells of the two matrices are determined by a comparison between $v_{i,j}$ and $h_{i,j}$:

$$M_{i,j} = \begin{cases} m_v & \text{if } v_{i,j} \leq h_{i,j} \\ m_h & \text{else} \end{cases} \qquad L_{i,j} = \begin{cases} l_v & \text{if } v_{i,j} \leq h_{i,j} \\ l_h & \text{else} \end{cases} \tag{2.8}$$

A visualization of how a cell of the dynamic programming matrices is filled using the above recurrence can be observed in Figure 2.3. The recurrence can be used to fill the two dynamic programming matrices row- or column-wise, because the value of a cell depends only on cells with strictly smaller indices. The trace-back can then be started from the cell $M_{x-1,y-1}$. The whole algorithm in this form can be implemented without need for any sophisticated data structures or sub-algorithms.

2.2.4 Analysis

The algorithm needs to keep the matrices in memory and therefore has a space usage in $O(x \cdot y)$. The running time is in $O(x \cdot y \cdot \max(x, y))$. Filling the cell $M_{i,j}$ of the matrices

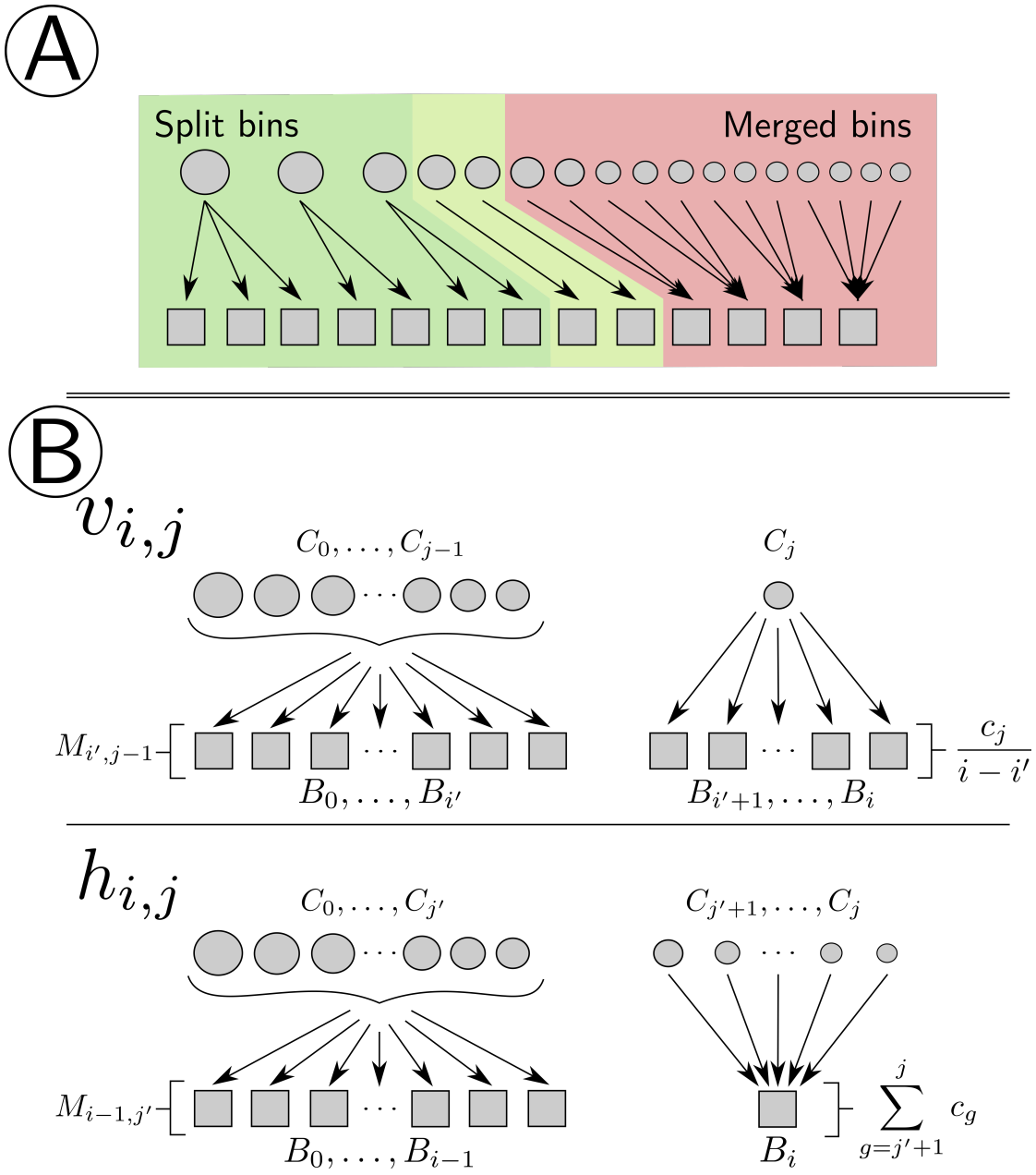


Figure 2.2: User bins are shown as circles and technical bins are shown as squares. At the top (A), there is an example of how a sequence of user bins could be split and merged into a sequence of technical bins. Below (B) is a sketch of the two formulas inside the minima from the main recurrence of the hierarchical binning algorithm (eqs. 2.5, 2.6).

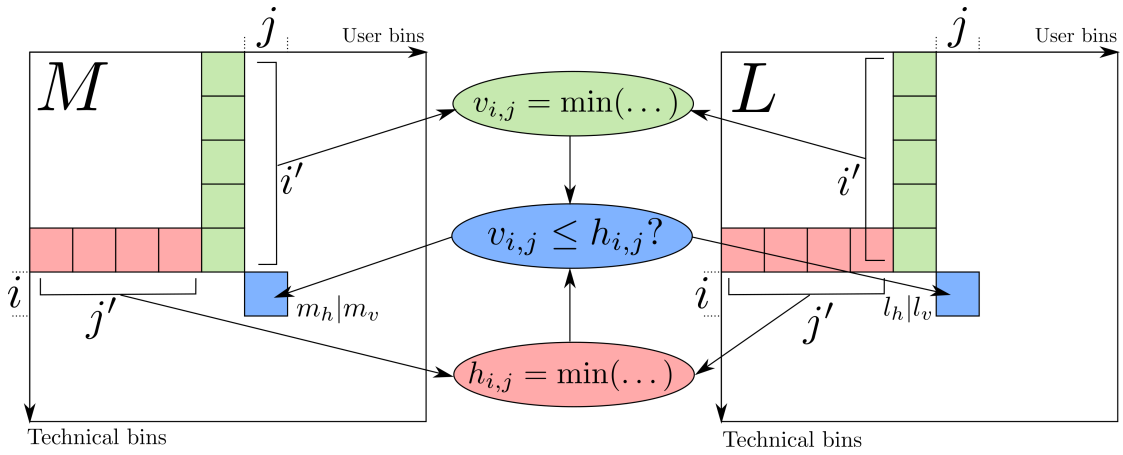


Figure 2.3: A visualization of how a value for a cell of the dynamic programming matrices M and L is computed using the main recurrence of the hierarchical binning algorithm (eqs. 2.5, 2.6). The current cell is marked with a blue color. Cells that contribute to the computation of $v_{i,j}$ are shown in green. Cells that contribute to the computation of $h_{i,j}$ are shown in red. The trace-back matrix is omitted to make the visualization more comprehensible.

requires computing minima over i and j many values. On average, $x/2$ and $y/2$ values are compared. The larger of the two dominates. The sums in the formula of $h_{i,j}$ (eq. 2.6) can be computed iteratively together with the minimum. Therefore, computing one value inside the minimum is in $O(1)$. In total, $x \cdot y$ cells have to be filled in $O(\max(x, y))$ time each.

In practice, x and y are generally small compared to the sizes of the actual sequences in the user bins. Therefore, computing the values $c_0, \dots, c_{y-1}$ might often be a more computationally intensive task than the hierarchical binning afterwards. Since they are needed for the algorithm, one could consider the running time $O(n + x \cdot y \cdot \max(x, y))$, where n is the sum of the k-mer multiset sizes of all user bins.

If the number of user bins y is very large, one can manage the running time of the hierarchical binning algorithm by allowing more recursive levels and reducing the number of allowed technical bins x in each level. Reducing x forces the algorithm do merge more user bins. These resulting merged technical can then be resolved on lower levels of the recursion.

2.3 Cardinality Estimates

2.3.1 Adjustments To The Hierarchical Binning

The previously described algorithm uses sums of the originating user bins' cardinalities to estimate the cardinalities of merged bins. Note that this refers to the initialization (eq. 2.4) of M and the term (III) (eq. 2.6) in the recursion, but not the initialization of

L and term (IV). The latter keep track of the amount of k-mers sent to the lower levels and do not refer to the merged bin on the current level.

Estimating the cardinality of merged bins this way can be a drastic overestimation. If the underlying sets of the merged bins have large intersections, the size of their union will be much smaller than the sum of their cardinalities. Using more accurate estimates could empower the algorithm to find a more space efficient arrangement of bins.

Let $U_{i,j}$ be an estimate of the cardinality of the k-mer multiset $C_i \cup C_{i+1} \cup \ldots \cup C_j$. In particular, $\forall i : U_{i,i} \approx c_i$. The hierarchical binning algorithm can me modified in the following way. The second part of the initialization becomes:

$$\forall j \in \{1, \ldots, y-1\} : M_{0,j} = U_{0,j},\ L_{0,j} = \alpha \cdot \sum_{g=0}^{j} c_g \tag{2.9}$$

This the case where all j user bins have to be merged into the single available technical bin. $U_{0,j}$ gives an estimate of the cardinality of this exact technical bin.

The only change to the main recurrence is in part (III):

$$h_{i,j} = \min_{j' \in \{0,\ldots,j-2\}^{(*)}} \left(\underbrace{\max(M_{i-1,j'}, U_{j'+1,j})}_{\text{(III)}} \cdot (i+1) + \alpha \cdot \underbrace{\left(L_{i-1,j'} + \sum_{g=j'+1}^{j} c_g \right)}_{\text{(IV)}} \right) \tag{2.10}$$

*only consider j' if the user bin $C_{j'}$ was not truly split in the intermediate result in the cell $M_{i-1,j'}$. (2.11)

The sum that estimated the cardinality of the new merged bin is again replaced by the corresponding look-up in the matrix U. The constraint for j' (eq. 2.11) is not guaranteed to preserve the final result anymore, because there could be some advantageous merges in the excluded intermediate results due to a high similarity of merged bins. The changes to the sorted order discussed in section 2.4 might amplify this problem. Whether to keep the constraint is now a trade-off between running time and quality of the result.

To compute U, HyperLogLog sketches can be used. First of all, they have to be built for every user bin. Afterwards, sketches have to be computed for all possible intervals of user bins in the sorted order via merging. More formally, $U_{i,j}$ must be defined $\forall\, 0 \leq i \leq j < y$. With careful implementation, roughly one merge is needed per computed value, see algorithm 1.

2.3.2 Analysis

The running time of algorithm 1 is in $O(n + y^2 \cdot m)$, where n is the sum of the k-mer *multiset* sizes of all user bins and m is the size of the HyperLogLog sketches. The first

Algorithm 1 Computation of estimated cardinalities for U

```
procedure COMPUTE_ESTIMATES(m, C_{0,...,y-1})
    for i ∈ {0, ..., y − 1} do                          ▷ compute sketch once per bin
        HLL_i ← HyperLogLog sketch of size m on C_i

    for i ∈ {0, ..., y − 1} do
        HLL_curr ← HLL_i                    ▷ start with sketch of first bin of the interval
        for j ∈ {i, ..., y − 1} do
            HLL_curr ← MERGE(HLL_curr, HLL_j)                    ▷ iteratively merge
            U_{i,j} ← ESTIMATE(HLL_curr)
```

for loop takes time in $O(n)$ to compute the HyperLogLog sketches. n can be *very* large in practice. But the whole preprocessing for the IBF including the hierarchical binning is in $O(n)$ anyway, since the values $c_0, \ldots, c_{y-1}$ can only be computed in this time.

In an implementation, one could use the estimates on the diagonal of U as the user bin cardinalities. Alternatively, they could be computed exactly. This is even more time and space demanding in practice.

The second **for** loop takes time in $O(y^2 \cdot m)$, because MERGE and ESTIMATE both have running times in $O(m)$. As mentioned in the introduction, these operations are very efficient. m can be set to values like 1024, 2048 or 4096 to achieve good relative errors (see 1.2) with a low running time impact.

The space usage of the algorithm is in $O(y \cdot \max(y, m))$, because a $y \times y$ matrix and an array of y m-sized HyperLogLog sketches are stored.

2.4 Arrangement By Similarity

2.4.1 Arrangement Definition

The first step of the hierarchical binning algorithm is to sort the user bins by their cardinality in decreasing order. This massively reduces the number of different merges the algorithm takes into account, because only intervals of the sorting are considered. Whereas in theory all exponentially many subsets of user bins could be merged, the number of intervals is in $O(y^2)$.

The decreasing order is a natural choice for this step in the original algorithm, since it is expected that mainly small bins are merged. These are close to each other in the sorted order. With the addition of cardinality estimates for merged bins from section 2.3, the requirements become more diverse.

It is more appropriate to talk about an *arrangement* rather than a sorting of the user bins from now on. There is no obvious order to be defined. It is still desirable that user bins of similar cardinality are close to each other in the arrangement, for the same reason as before. But it is also important that user bins of similar k-mer content are close to each other. In other words, the larger the intersection of bins are, the closer

they should be.

With such an arrangement of user bins in combination with the cardinality estimates, the algorithm can better detect advantageous merge patterns. Merging many small and similar user bins often frees up technical bins that can be used for splitting larger user bins. The merged bins themselves are very space efficient on the current level, because of the large intersections between the originating user bins.

Defining the desired arrangement more rigorously and unambiguously is hard, because it is difficult to predict which exact merges will lead the algorithm to the best result. The given definition is sufficient for the purposes of this work. The following algorithm 2 employs a simple agglomerative hierarchical clustering heuristic to group similar bins together before the cardinality estimates are computed on the fixed user bin arrangement.

2.4.2 Clustering Algorithm

The algorithm inputs are the decreasingly sorted user bin cardinalities $c_0, \ldots, c_{y-1}$, the respective HyperLogLog sketches and a real number $0 \leq r \leq 1$. The output is a permutation of the user bins which represents the arrangement described above.

In the first step of the algorithm, the sequence is split into intervals (algorithm 2, ARRANGE_IN_INTERVALS). These intervals are chosen such that they are as large as possible while the cardinality of the largest bin in the interval times r is still smaller than the cardinality of the smallest bin. The clustering is performed on each of these intervals independently. This way, the sequence globally remains roughly sorted by bin cardinality, because only the positions of bins of similar cardinality are swapped in comparison to the original sorted sequence. The parameter r influences how large the intervals are and how finely grained the original sorting remains.

The local rearrangement inside the intervals is extracted from a clustering tree. This tree is computed with a simple agglomerative hierarchical clustering scheme (algorithm 2, CLUSTER). The Jaccard distances of user bin k-mer sets are estimated using the HyperLogLog sketches and serve as distances for the clustering (algorithm 2, JACCARD).

In the initial state of the algorithm, each user bin of the current interval represents a single cluster, i.e. a leave in the tree. Then, the two clusters with minimal distance are joined and the distance matrix D is updated. This procedure is repeated until there is only one cluster left. The joining is realized by connecting the roots of the old clusters with a new node. A HyperLogLog sketch for the new cluster is computed by merging the sketches of the originating clusters. The distances from the new cluster to previously existing ones are computed by estimating the Jaccard distance as before.

The final cluster is a binary tree where the leaves represent user bins. It roughly holds that the shorter the path between two leaves is, the more similar they are. Therefore, any depth-first traversal of the tree that only extracts the leaves produces a user bin arrangement as desired, for the current interval. Only at the borders between two intervals user bins are placed right next to each other without having been clustered together.

Note that this property remains unchanged when a sub-tree of any clustering tree is rotated along its root. Rotating here means that the left and right sub-trees of the root

are swapped. It follows that the arrangement implied by the tree is ambiguous, because rotations would alter the arrangement while preserving proximity of similar bins. This freedom is used to tackle the problem that the bins at the borders between intervals are not clustered together.

The solution is to link the clusterings for two neighboring intervals by an overlap bin. This overlap bin is chosen iteratively as the rightmost bin from the previous interval. After the clustering of the current interval, the overlap bin is rotated to the left and ignored during the traversal of the tree. This way, the intervals are at least linked by a single user bin that influences the clustering on both sides.

A visualisation of the clustering in intervals can be seen in Figure 2.4.

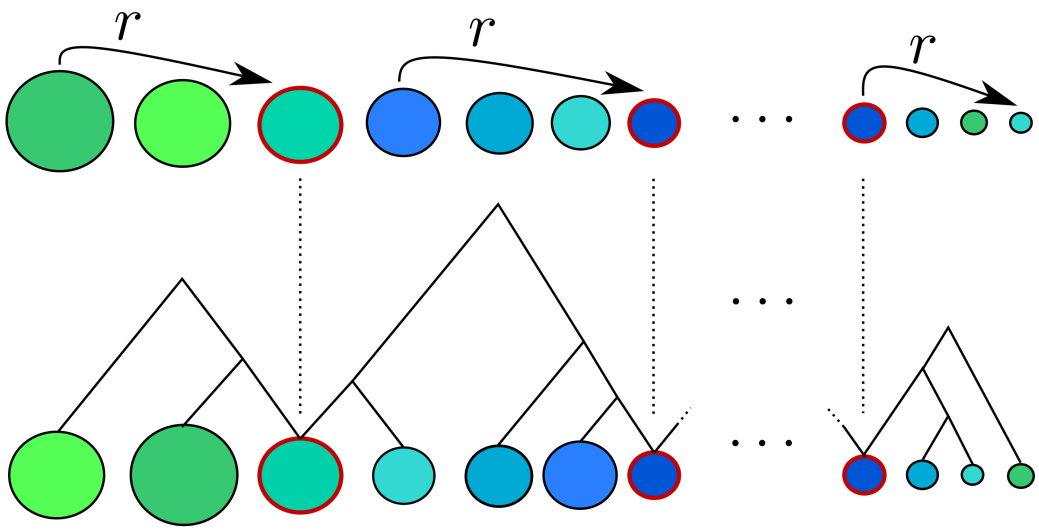


Figure 2.4: A visualisation of the clustering in intervals. User bins are shown as circles. Similar colors indicate similar k-mer content. After the clustering, similar colors are closer to each other and on a global scope, the sequence is still decreasingly sorted. Only a few local rearrangements have been done. Overlap bins are marked by a red border. Their positions are fixed, because they appear in two subsequent clustering trees.

In combination with the previously described cardinality estimates, the whole adjusted hierarchical binning is efficiently realized with the following sequence of steps:

1. Build the HyperLogLog sketches on the user bin k-mer multisets and estimate the single bins' cardinalities. Alternatively, compute these cardinalities in exact fashion and only use the HyperLogLog sketches for the step 4.

2. Sort user bins decreasingly by their cardinalities.

3. Compute better arrangement by clustering of intervals on the sorting with algorithm 2.

4. Estimate cardinalities of all possible merged bins regarding the arrangement by merging the already computed HyperLogLog sketches with algorithm 1.

5. Compute adjusted hierarchical binning.

2.4.3 Analysis

Let l denote the number of bins in a single run of the clustering algorithm (algorithm 2, CLUSTER). The dominating steps of the algorithm are the initialization of the distance matrix D and the while-loop. The traversal of the tree is not relevant for the running time, because there are only roughly $2 \cdot l$ nodes in the tree.

Since the number of clusters decreases by one in every iteration, the number of iterations in the while-loop is $l - 1$. The running time of a single iteration depends on the implementation of the distance matrix. An implementation based on arrays or hashmaps would result in a total running time in $\Omega(l^3)$. In every iteration of the while-loop the minimum of up to $\Omega(l^2)$ unordered values would have to be to be computed. An implementation with a heap-based priority queues is more efficient. It is assumed in the following that the priority queue is based on a standard binary heap. More sophisticated heaps would not improve the total running time of the algorithm.

The initialization of D takes time in $O(l^2 \cdot \log l)$, because the number of values to be inserted is in $O(l^2)$.

The running time of an iteration of the while-loop is in $O(l \cdot \max(\log l, m))$, where m is the size of the HyperLogLog sketches. Finding the minimum of D in one iteration is in $O(1)$. To update D in one iteration, at most l values have to be removed. Then, at most l new distances have to be computed in $O(m)$ time and inserted into the priority queue. One whole updating step therefore has a running time in $O(l \cdot \max(\log l, m)))$.

In summary, a whole clustering takes time in $O(l^2 \cdot \max(\log l, m)))$ to compute. It follows that the running time of the whole arrangement finding algorithm is in $O(y^2 \cdot \max(\log y, m)))$. This running time is especially realized when r is close to 0 and all user bins are clustered in a single clustering run. When r tends to 1, the running time instead tends to something in $O(y \cdot m)$, because there will be roughly y clustering steps, each with a very small l.

The space requirements are again in $O(y \cdot \max(y, m))$. The distance matrix needs to store $O(y^2)$ values and there are at most $2 \cdot y$ HyperLogLog sketches of size m in memory at the same time.

Algorithm 2 Arrangement by clustering of intervals

procedure ARRANGE_IN_INTERVALS($c_{0,\dots,y-1}$, $HLL_{0,\dots,y-1}, r$)
  $a \leftarrow$ empty arrangement
  $first, last \leftarrow 0, 1$

  while $first < y$ **do**
    if $c_{first} \cdot r > c_{last}$ **or** $last == y$ **then** ▷ Next bin too small for this interval
      CLUSTER($a, c_{first,\dots,last-1}, HLL_{0,\dots,y-1}$)
      $first \leftarrow last$
    $last \leftarrow last + 1$
  return a

procedure CLUSTER($a, c_{first,\dots,last-1}, HLL_{0,\dots,y-1}$)
  $clusters \leftarrow \{first, \dots, last - 1\}$
  if a is **not** empty **then** ▷ Add one overlap bin to this interval
    $c_x \leftarrow$ last element from a
    $clusters \leftarrow clusters \cup \{x\}$

  for $\{i, j\} \subseteq clusters$ **do**
    $D_{i,j} \leftarrow$ JACCARD(HLL_i, HLL_j) ▷ Initialize distance matrix

  while $|clusters| > 1$ **do**
    $i, j \leftarrow$ clusters with minimal $D_{i,j}$
    $u \leftarrow (i, j)$ ▷ Join clusters
    $HLL_u \leftarrow$ MERGE(HLL_i, HLL_j)

    Remove i and j from D and $clusters$ ▷ Update D
    for $k \in clusters$ **do**
      $D_{u,k} \leftarrow$ JACCARD(HLL_u, HLL_k)
    $clusters \leftarrow clusters \cup \{u\}$

  $tree \leftarrow$ last element in $clusters$
  if a is **not** empty **then**
    Rotate $tree$ such that x is the leftmost leaf ▷ Move overlap bin to the left
    Delete x from $tree$

  for i in DFS traversal of $tree$ **do**
    if i is a leaf **then**
      Append c_i to a ▷ Extract local rearrangement

procedure JACCARD(HLL_a, HLL_a)
  $HLL_{a,b} \leftarrow$ MERGE(HLL_a, HLL_b)
  return $2 - (\text{ESTIMATE}(HLL_a) + \text{ESTIMATE}(HLL_b)) \,/\, \text{ESTIMATE}(HLL_{a,b})$

3 Results

The proposed changes to the hierarchical binning algorithm were tested with different parameters on multiple data sets. All benchmarks were performed on a machine with 387 GB of main memory and an Intel® Xeon® CPU E5-2667 v2 @ 3.30GHz processor with 32 logical cores. 16 Threads were used in all parallel sections. The data sets were stored on an SSD storage device.

3.1 Constructed Data Set

The first data set was artificially constructed to show the potential benefits of the improvements to the hierarchical binning proposed by this work. The structure of the data set is unfavorable for the algorithm in its original form. The improved algorithm should be able to find a better solution that takes advantage of the sequence similarities in the data set. It is not expected that the improvements have such strong effects on any real data sets.

The data set consists of one randomly generated DNA sequence of length 200 kilobases and 3 shorter randomly generated DNA sequences of length 10 kilobases. Additionally, for each of the smaller sequences there are 20 very similar sequences. These sequences were generated using the mason variator [9] on the originating small sequences with a SNP rate of 0.001 and a small indel rate of 0.0001.

The benchmarks were performed in the following way. Every sequence of the data set was registered as a user bin. The cardinalities of the bins were computed exactly and HLL sketches were built. The hierarchical binning was run three times with 16 technical bins. Once in its original form (*Reference*), once with the cardinality estimates from section 2.3 (*Estimates*) and once with both the cardinality estimates and the arrangement by similarity from section 2.4 (*Arrangement*). The parameter r for the rearrangement was set to 0.5. The influence of this parameter was examined later in detail for the real data set. The parameter α of the hierarchical binning was set to 1.2. This value can be interpreted as the expectation of a small IBF overhead on lower levels.

Please note that these benchmarks target only the hierarchical binning on the highest recursive level. No further recursive steps were done to acquire a complete blueprint for a hierarchical IBF. The only available result is the transformation from user bins to technical bins (IBF blueprint) on the highest level of the hierarchical IBF and the number of k-mers in merged bins that have to be resolved on lower levels. The quality of the proposed changes can therefore only be assessed using these metrics. To evaluate how the different algorithmic parameters influence the actual IBF size with multiple recursive levels is not in the scope of this work.

A quantitative comparison of the results of the different runs can be observed in Figure 3.1. Using cardinality estimates reduces the number of merged bins in comparison the the original algorithm. This is generally favorable, because merged bins lead to a running time overhead. Additionally, the S_{tech} of the top level IBF and the number of k-mers sent to lower level via the recursion are reduced. Also rearranging the user bins leads to a very compact top level IBF and moderately more lower level k-mers. Due to the space efficient top level IBF, the result of the algorithm that uses both adjustments can be considered the best.

A visualization of the actual transformations from user bins to technical bins can be seen in Figure 3.2. For the reference algorithm it can be observed that all merged bins have a large IBF overhead. This is likely due to the overestimation of the bin cardinality by the sum of user bin cardinalities. Using cardinality estimates allows the algorithm to detect the actual cardinalites of the technical bins. This leads to a better result and more technical bins can be used to split the user bin of the long sequence. The problem is that this algorithm is still forced to merge small user bins of the different types, because the sorted order of the user bins does not take into account the sequence similarity and the algorithm can only merge adjacent intervals of the sequence. The algorithm that also rearranges the bins finds the space efficient result that was expected when constructing the data set.

3.2 Real Data Sets

The proposed adjustments of this work were tested on a real and practically relevant data set. The data set consists of all complete genomes of archaea and bacteria species in the NCBI RefSeq data base [10]. This pan-genome could be used in practice for metagenomic analysis of microorganisms.

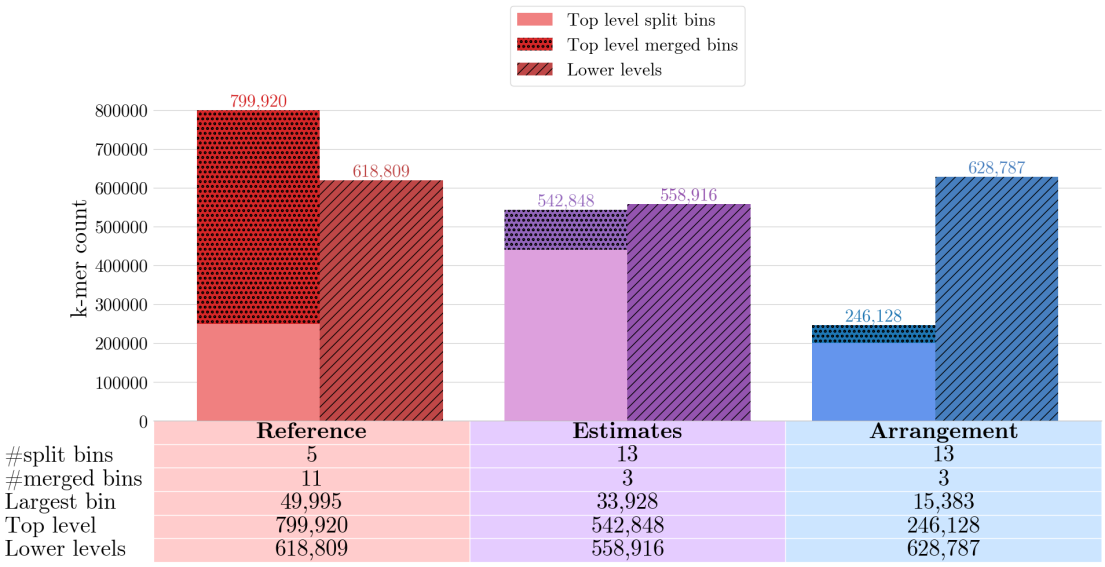
First, the data set was used in its complete form (section 3.2.1). It consists of roughly 22000 genomes. Its total compressed size is roughly 25 GB and its total decompressed size is roughly 100 GB.

Afterwards, the algorithms were tested on the same data set with a filter that allows only one genome per species (section 3.2.2). This reduced data set consists of roughly 6200 genomes. Its compressed size is roughly 6 GB and its decompressed size is roughly 24 GB. It is expected that the adjustments have smaller effects on this second data set, because the filter removes most or all of the clusters of highly similar sequences in the data set.

For both of these data sets, the benchmarks were performed in a similar way as for the constructed data set. Every genome was registered as a user bin and the three versions of the hierarchical binning were performed only for the top level. The parameter α of the hierarchical binning was again set to 1.2.

3.2.1 All Complete Genomes

Multiple sets of benchmarks were performed on this data set to investigate how the different parameters of the algorithms influence the result quality, running time and



	Reference	Estimates	Arrangement
#split bins	5	13	13
#merged bins	11	3	3
Largest bin	49,995	33,928	15,383
Top level	799,920	542,848	246,128
Lower levels	618,809	558,916	628,787

Figure 3.1: A quantitative comparison of the results of the three versions of the hierarchical binning on the constructed data set. For every column, the left bar of the bar plot represents the top level IBF for which the hierarchical binning computed an exact assignment of user to technical bins. The height of the bar is the S_{tech} of this IBF. Split bins have light colors and merged bins are represented by a dotted pattern. The right bar has a hatched pattern and represents the number of distinct k-mers in merged bins that are sent to lower recursive levels. Please note that the height of this bar is not necessarily equivalent to the sum of the S_{tech} values for all lower level IBFs. The IBF overhead is not accounted for, because the hierarchical binning was only computed for the top level. The term "Top Level" refers to the S_{tech} of the top level IBF and the term "Lower Levels" refers to the number of k-mers sent to lower levels.

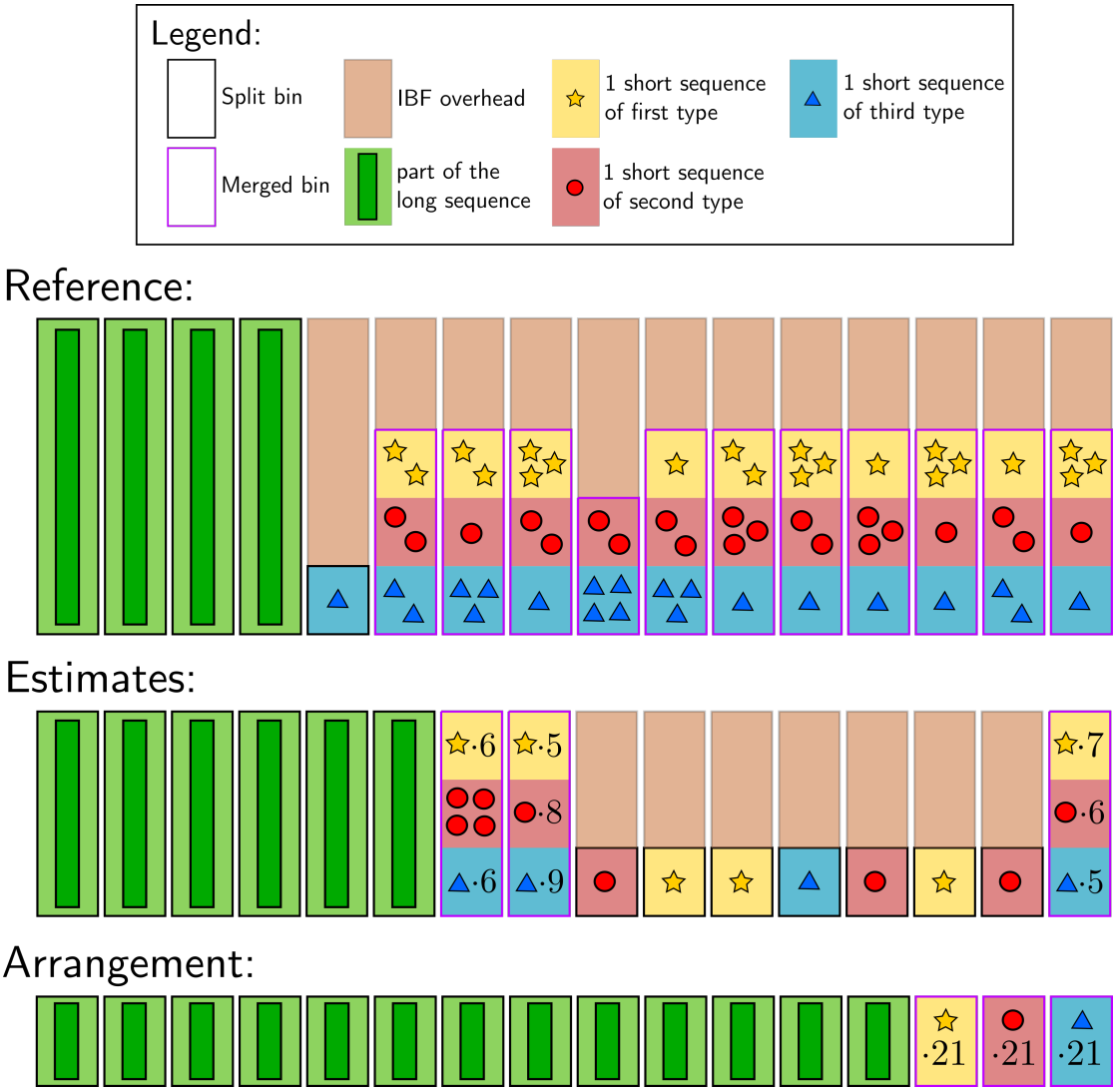


Figure 3.2: A visualization of the different transformations from user bins to technical bins that the three runs of the hierarchical binning produced on the constructed data set. For each of the transformations, the 16 technical bins of the top level IBF are shown as rectangles. The shapes inside the technical bins indicate the user bins that are contained in the technical bin. A shape times a number n means that n user bins of this type are contained in the technical bin.

space usage.

The first set of benchmarks focused on the number of technical bins. In practice, this number is usually set to one of the powers of two around 1024. Smaller numbers make more levels in the hierarchical IBF necessary. Larger numbers make the *contains* query of the IBFs slow. A detailed comparison of the quantitative results of the different versions of the hierarchical binning with 1024 technical bins can be observed in Figure 3.3.

For 1024 technical bins, the results of all three algorithms contain almost exclusively merged bins. This is expected, because the number of user bins (roughly 22000) is much larger than the number of technical bins. The number of k-mers sent to lower recursive levels is almost equivalent for all three results, because almost all technical bins are merged bins. The main difference between the three algorithms lies in the S_{tech} of the top level IBF. Using cardinality estimates reduces the S_{tech} of the top level IBF by roughly 25% in comparison the the original algorithm. Using cardinality estimates and rearranging the user bin sequence reduces the S_{tech} of the top level IBF by roughly 60% in comparison the the original algorithm.

An overview of how the number of technical bins changes the result, running time and peak memory usage can be observed in Figure 3.4. The parameter r of the clustering in intervals was set to 0.0 in all of the runs. This leads to all user bins being clustered in a single run of the clustering algorithm.

The running time grows slightly faster than linear in terms of the number of technical bins for all versions of the hierarchical binning. For larger numbers of technical bins, the running times of the three compared algorithms are very similar. It follows that the running time is dominated by the dynamic programming step of the original hierarchical binning in these cases.

The peak memory usage of the algorithm with cardinality estimates and user bin rearrangement is significantly higher than the peak memory usage of the original algorithm in all cases. This is due to the cardinality estimates matrix U and the data structures involved in the clustering. For larger numbers of technical bins, the original algorithm has a relatively higher memory usage. This is expected, because the number of technical bins does not influence running time and memory usage of the cardinality estimation algorithm or the clustering algorithm.

In all cases, the algorithm with both adjustments produces a more compact top level IBF blueprint and sends a smaller or equal number of k-mers to lower levels in comparison to the original algorithm. The results for smaller numbers of technical bins appear better, but the merged bins might be easier to resolve on lower levels when they contain less user user bins. This is generally the case for higher number of technical bins.

The second set of benchmarks focused on the parameter r of the clustering in intervals. The different results are compared in Figure 3.5. The number of technical bins was set to 1024 in all runs. Only the algorithm that rearranges the user bin sequence is influenced by the parameter r. The closer r is to 1, the smaller the running time becomes and the closer it is to the running times of the original algorithm and the algorithm that uses only the cardinality estimates. For the value $r = 0.0$, the single large clustering

takes up roughly half of the running time of the whole algorithm. For the value $r = 0.9$ the many smaller clusterings take up only a small part of the running time. For the value $r = 0.0$, the algorithm with the user bin rearrangement has a much higher peak memory usage than the algorithm that uses only cardinality estimates. For $r \geq 0.25$, the peak memory usage is close to the peak memory usage of the algorithm that uses only cardinality estimates. The result qualities for the different values of r are very similar. Values closer to 1 lead to a very slightly larger S_{tech} of the top level IBF.

Further benchmarks were performed to investigate the effect of removing the optimization that constraints the range of values j' can take in the recursion of the hierarchical binning (eq. 2.11). The results were equivalent in most cases and otherwise had relative differences of less than 0.01% regarding the k-mer counts on the top and lower levels.

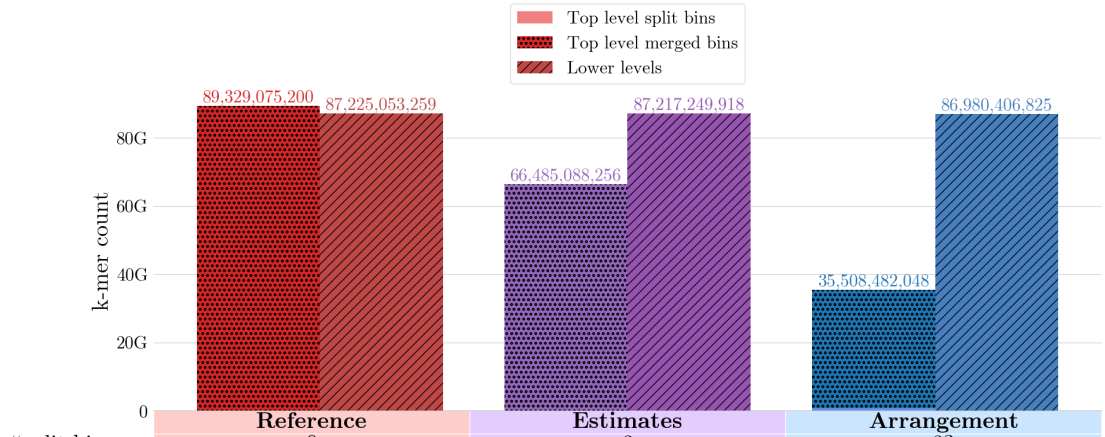
Finally, it was tested how the results of the algorithms change when no exact cardinalities for single uses bins are computed and HLL estimates are used instead. The parameter r was set to 0.0 and the number of technical bins was set to 1024. The results of the original algorithm and the algorithm that uses both cardinality estimates and rearranges the user bin sequence did not change in comparison to the previous runs where exact cardinalities were used. The result for the algorithm that uses only cardinality estimates had a less than 2% larger top level IBF and less than 0.01% more k-mers sent to lower levels in comparison to the corresponding previous runs with exact cardinalities.

It follows that omitting the computation of exact user bin cardinalities could be a reasonable approach for performance-critical applications. Figure 3.6 compares the running times and peak memory usages of the different cardinality computation algorithms (for single user bins) and the different versions of the hierarchical binning. These two algorithmic steps are viewed as a pipeline. For the hierarchical binning runs, the number of technical bins was set to 1024 and the parameter r was set to 0.0. For these parameters, the running time of the whole pipeline was dominated by the exact cardinality computation using an optimized hash table [11]. Increasing the number of technical bins would make the hierarchical binning use most of the total running time of the pipeline. The version of the hierarchical binning that uses cardinality estimates and rearranges the user bin sequence had the highest peak memory usage. This memory usage could be reduced by increasing the parameter r (see figure 3.5).

3.2.2 One Genome Per Species

For this reduced data set, a single set of benchmarks was performed to investigate how the adjusted versions of the hierarchical binning perform on a data set with less or no clusters of highly similar sequences. The experiments were conducted in a similar way to the experiments in the previous sections. The parameter α was again set to 1.2 and the parameter r was set to 0.0.

A detailed overview of the quantitative results for the run with 1024 technical bins can be observed in Figure 3.7. The result of the original algorithm contains almost exclusively merged bins on the top level. The results of the adjusted algorithms contain small fractions of split bins on the top level. More split bins than in the case of the complete real data set are expected, because the number of user bins (6200) is smaller. Therefore,



	Reference	Estimates	Arrangement
#split bins	0	2	23
#merged bins	1,024	1,022	1,001
Largest bin	87,235,425	64,926,844	34,676,252
Top level	89,329,075,200	66,485,088,256	35,508,482,048
Lower levels	87,225,053,259	87,217,249,918	86,980,406,825

Figure 3.3: A visualization of the results of the three versions of the hierarchical binning with 1024 technical bins on the complete real data set. The term "Top Level" refers to the S_{tech} of the top level IBF and the term "Lower Levels" refers to the number of k-mers sent to lower levels.

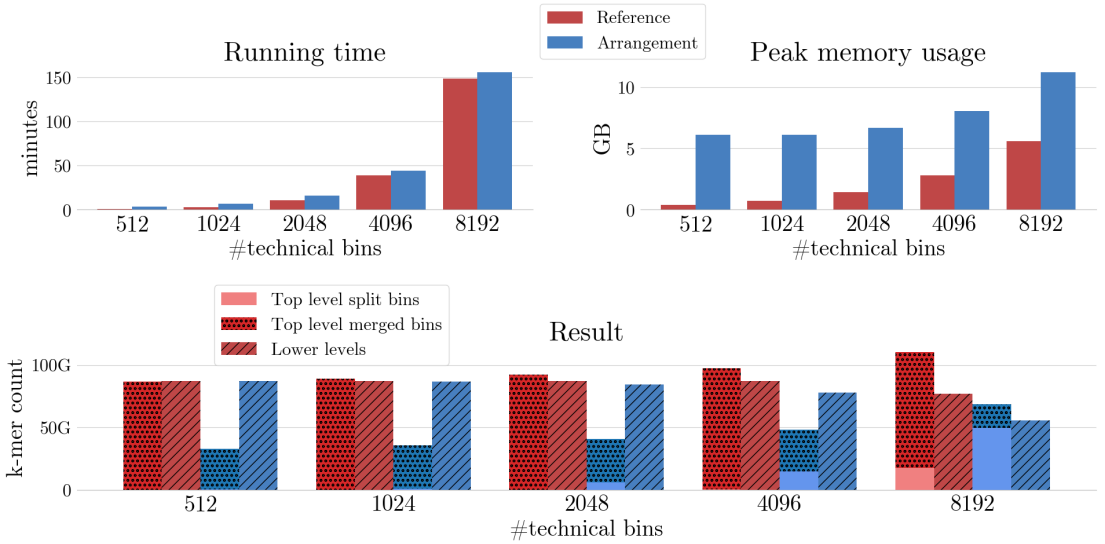


Figure 3.4: An overview of how the number of technical bins changes the result, running time and peak memory usage on the complete real data set. Only the reference algorithm (red) and the algorithm with cardinality estimates and user bin rearrangement (blue) are compared to make the plot more comprehensible.

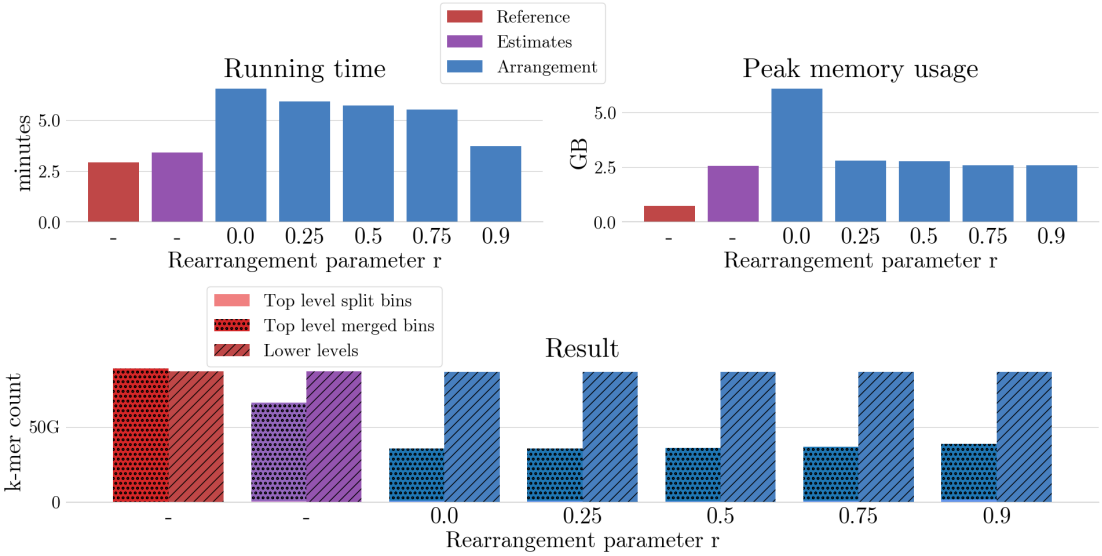


Figure 3.5: A visualization of how the parameter r of the clustering in intervals influences the running time, peak memory usage and result quality (blue). The original algorithm (red) and the version that only uses cardinality estimates (purple) are not influenced by this parameter.

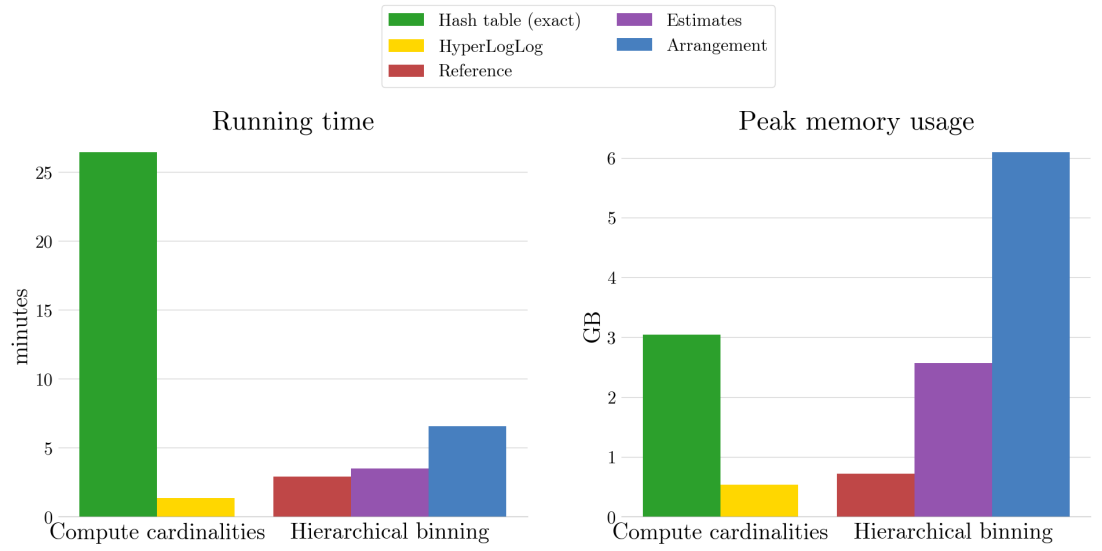
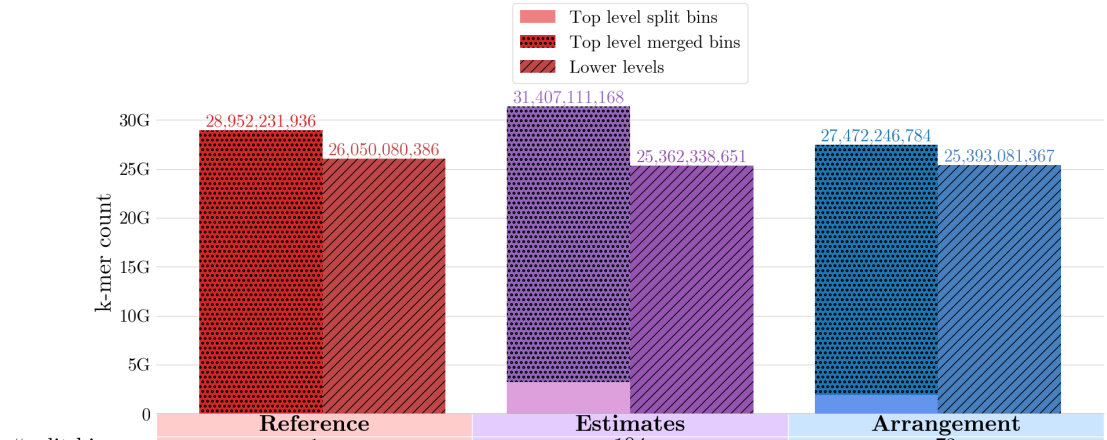


Figure 3.6: A comparison of the running times and peak memory usages of the two different ways of computing the cardinalities of single user bins and the three versions of the hierarchical binning.



	Reference	Estimates	Arrangement
#split bins	1	104	72
#merged bins	1,023	920	952
Largest bin	28,273,664	30,671,007	26,828,366
Top level	28,952,231,936	31,407,111,168	27,472,246,784
Lower levels	26,050,080,386	25,362,338,651	25,393,081,367

Figure 3.7: A visualization of the results of the three versions of the hierarchical binning on the reduced data set. The term "Top Level" refers to the S_{tech} of the top level IBF and the term "Lower Levels" refers to the number of k-mers sent to lower levels.

the sum of k-mers sent to lower levels is slightly smaller for the adjusted algorithms. The result of the algorithm that uses only cardinality estimates has the largest S_{tech} on the top level. The algorithm that uses cardinality estimates and rearranges the user bin sequence (blue) performed slightly better than the original algorithm in this category.

Figure 3.8 compares the results, running times and peak memory usages of different runs with varying numbers of technical bins. The trends of the running time and peak memory usage are similar to the trends observed on the complete data set. The result quality of the reference and the adjusted algorithm is close to equivalent in all runs. The adjusted algorithm seems to perform slightly better in most cases. An exception is the run with 2048 technical bins.

3.3 Implementation

An implementation of the proposed adjustments from the sections 2.3 and 2.4 using the SeqAn library [12] can be found under `https://github.com/Felix-Droop/Chopper/tree/v1.0/include/chopper` in the two directories `union` and `pack`. This implementation was used for the benchmarks presented in the previous sections and is based on this original implementation by Svenja Mehringer: `https://github.com/seqan/chopper/commit/b0e880de972c9f09cdea2ab1135f46ec883cda4b`.

In the added parts of the implementation, the cardinality estimation algorithm and the clustering in intervals are both parallelized with threads. Certain running time crit-

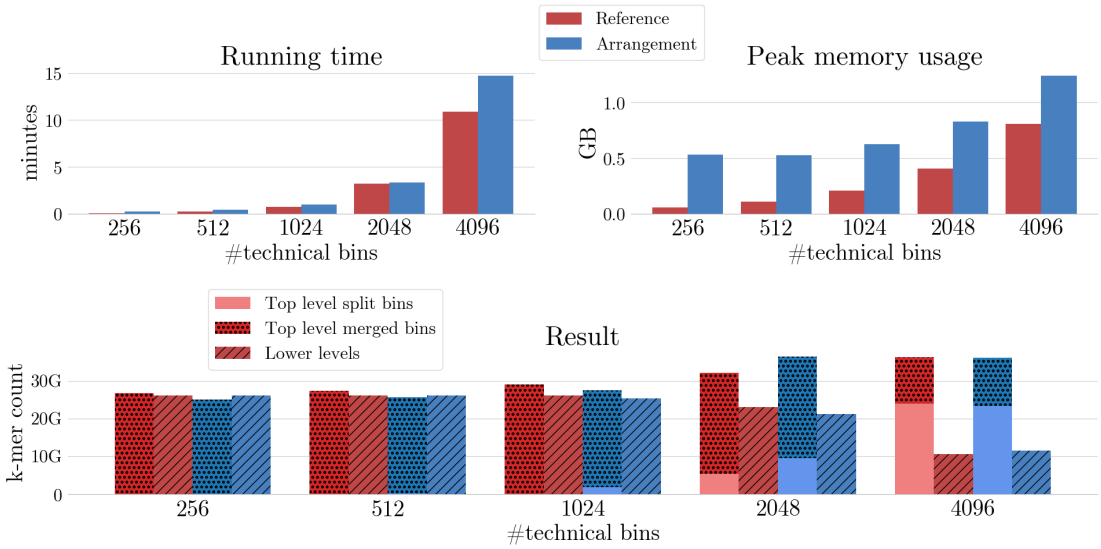


Figure 3.8: An overview of how the number of technical bins changes the result, running time and peak memory usage on the reduced data set. Only the reference algorithm (red) and the algorithm with cardinality estimates and user bin rearrangement (blue) are compared to make the plot more comprehensible.

ical sections of the code like the HyperLogLog operations were additionally optimized using low-level optimization techniques such as SIMD instructions. The HyperLogLog implementation is based on an implementation by Hideaki Ohno [13]. Multiple performance improving adjustments were made to this implementation, including the usage of a faster hash function [14].

All scripts used to produce the benchmarks of the previous sections are available at https://github.com/Felix-Droop/BscScripts/tree/v1.0 for easy reproducibility.

4 Discussion

4.1 Interpretation Of Results

The experiments on the constructed data set and the complete real data set show that the proposed adjustments to the hierarchical binning can lead to large reductions in the S_{tech} of the top level IBF. The analysis of the results on the constructed data set (Figure 3.2) shows how the adjusted algorithm is capable of finding more advantageous merging patterns.

On the reduced real data set with only one genome per species, the adjusted algorithm seems to perform about as good as the original algorithm. When looking at the benchmark data (Figure 3.8), it seems like the adjusted algorithm performs slightly better for smaller numbers of technical bins and slightly worse for larger numbers. However, these tendencies could be just random fluctuations.

Generally, the results have to be evaluated with care. The recursion for the merged bins was not completed and the total size of the final hierarchical IBF is unknown. It was also not yet investigated how severely the cascading IBF queries due to many merged bins impact the running time. More split bins are very likely preferable and could represent a reasonable trade-off in comparison to a slightly more space efficient solution.

It appears that the adjusted algorithm needs clusters of highly similar sequences in the input data set to produce clearly better results than the original algorithm. The sequence similarity level between genomes of multiple organisms of the same species seems to be high enough, as can be seen in the results for the complete real data set. The sequence similarity levels between genomes of organisms of related species seems to be too low, as can be seen in the results for the reduced data set.

The benchmarks that tested different numbers of technical bins suggest that using smaller numbers of technical bins is the best strategy both in terms of running time, memory usage and result quality, i.e. expected memory usage of the resulting IBF. As mentioned, larger numbers of technical bins additionally make the IBF queries slow. Definite claims can be made in these regards once the implications of different numbers of technical bins have been examined for a complete multilevel IBF.

The benchmarks that focused on the parameter r of the clustering in intervals showed that the parameter has a small effect on the result. The running time and memory usage are greatly influenced. Larger values of r lead to very slightly worse results and significantly lower running times and memory usages. The mechanism of performing the clusterings in (almost) distinct intervals was originally intended to improve the result by keeping the user bin sequence globally roughly sorted by cardinality. This does not seem to work. Setting r to 0.0 and clustering all user bins together leads to the best

result.

Instead, the clustering in intervals controlled by the parameter r can serve as an effective way of reducing the running time and memory usage. It has a minor negative effect on the result. Most applications that work with very large data sets will happily accept this trade-off.

The used implementation was able to run on the complete real data set with a reasonable running time and peak memory usage. Approximating the user bin cardinalities with HLL sketches and computing the hierarchical binning with both adjustments with 1024 technical bins and $r = 0.9$ on 16 threads took roughly 5 minutes with a peak memory usage of less than 3 GB.

For even larger data sets, the number of technical bins can be further reduced at the cost of more recursive levels. The parameter r can be set close to 1 to minimize the performance impact of the user bin rearrangement. The memory usage of the cardinality estimates matrix could pose a problem for very large data sets, because it is asymptotically quadratic in the number of user bins. A sparse matrix representation or other compression techniques could be applied here. The running time of the whole program except the dynamic programming step can be reduced by using more threads on the hardware level. It is therefore expected that the given implementation will be able to handle most currently existing use cases with few or no adjustments using suitable hardware.

4.2 Conclusion

The hierarchical binning algorithm was introduced as a rebalancing procedure for unevenly sized sequence bin input of the Interleaved Bloom Filter (IBF). It is applied recursively to compensate for the internal merging of sequence bins. A hierarchical IBF containing IBFs on multiple levels is the result.

The proposed adjustments of this work try to improve the hierarchical binning such that it takes into account the sequence similarity of the bins and finds more advantageous merging patterns. The adjusted algorithm finds clearly better results on a real data set that contains multiple genomes per species. The top level IBF blueprint found by the adjusted algorithm has a 60% smaller memory consumption than the reference. The adjusted algorithm performs about as good as the original algorithm on a real data set that contains only a single genome per species. In total, the adjustments should be viewed as clear improvements to the algorithm. An optimized and practically applicable implementation is delivered.

4.3 Outlook

It remains to be seen how the adjusted hierarchical binning performs when a complete blueprint for a hierarchical IBF is computed using multiple recursive levels. Future work will have to investigate how this hierarchical IBF performs in real applications on possibly extremely large data sets.

There are a few algorithmic tweaks that could improve the results and were not yet discussed or implemented. The historic inverse probability (HIP) estimator [15] could improve the HLL cardinality estimates for single user bins. The HyperMinHash algorithm [16] could replace the HLL to improve the Jaccard distances in the clustering. More sophisticated clustering methods than the simple agglomerative clustering heuristic described in this paper exist and could produce a more desirable user bin arrangement.

Bibliography

[1] National Center for Biotechnology Information. *GenBank and WGS Statistics*. URL: https://www.ncbi.nlm.nih.gov/genbank/statistics/.

[2] T. Marschall et al. "Computational pan-genomics: status, promises and challenges". In: *Brief Bioinform* 19.1 (Jan. 2018), pp. 118–135. URL: https://doi.org/10.1093/bib/bbw089.

[3] Enrico Seiler et al. "Raptor: A fast and space-efficient pre-filter for querying very large collections of nucleotide sequences". In: *bioRxiv* (2020). DOI: 10.1101/2020.10.08.330985. eprint: https://www.biorxiv.org/content/early/2020/10/08/2020.10.08.330985.full.pdf. URL: https://www.biorxiv.org/content/early/2020/10/08/2020.10.08.330985.

[4] Vitor C. Piro et al. "ganon: continuously up-to-date with database growth for precise short read classification in metagenomics". In: *bioRxiv* (2018). DOI: 10.1101/406017. eprint: https://www.biorxiv.org/content/early/2018/08/31/406017.full.pdf. URL: https://www.biorxiv.org/content/early/2018/08/31/406017.

[5] Temesgen Hailemariam Dadi et al. "DREAM-Yara: an exact read mapper for very large databases with short update time". In: *Bioinformatics* 34.17 (Sept. 2018), pp. i766–i772. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty567. eprint: https://academic.oup.com/bioinformatics/article-pdf/34/17/i766/25702473/bty567.pdf. URL: https://doi.org/10.1093/bioinformatics/bty567.

[6] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

[7] Svenja Mehringer and Knut Reinert. *Hierarchical binning algorithm*. Personal communication. Nov. 2020.

[8] Philippe Flajolet et al. "HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm". In: (Mar. 2012). URL: http://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf.

[9] M. Holtgrewe. "Mason - A Read Simulator for Second Generation Sequencing Data". In: *Technical Report FU Berlin* (Oct. 2010). URL: http://publications.imp.fu-berlin.de/962/.

[10] Nuala A. O'Leary et al. "Reference sequence (RefSeq) database at NCBI: current status, taxonomic expansion, and functional annotation". In: *Nucleic Acids Research* 44.D1 (Nov. 2015), pp. D733–D745. ISSN: 0305-1048. DOI: 10.1093/nar/gkv1189. eprint: https://academic.oup.com/nar/article-pdf/44/D1/D733/9482930/gkv1189.pdf. URL: https://doi.org/10.1093/nar/gkv1189.

[11] Martin Ankerl. *robin_hood unordered map & set*. URL: https://github.com/martinus/robin-hood-hashing/releases/tag/3.11.1.

[12] Knut Reinert et al. "The SeqAn C++ template library for efficient sequence analysis: A resource for programmers". In: *Journal of Biotechnology* 261 (Nov. 2017), pp. 157–168. URL: http://publications.imp.fu-berlin.de/2103/.

[13] Hideaki Ohno. *HyperLogLog*. URL: https://github.com/hideo55/cpp-HyperLogLog/commit/517598b2fe3149291b007626f65191b95f750108.

[14] Yann Collet. *xxHash - Extremely fast hash algorithm*. URL: https://github.com/Cyan4973/xxHash/releases/tag/v0.7.3.

[15] Edith Cohen. *All-Distances Sketches, Revisited: HIP Estimators for Massive Graphs Analysis*. 2015. arXiv: 1306.3284 [cs.DS].

[16] Yun William Yu and Griffin M. Weber. *HyperMinHash: MinHash in LogLog space*. 2019. arXiv: 1710.08436 [cs.DS].