UNIVERSITÀ DEGLI STUDI DI VERONA
Department of Biotechnology

Master's degree in Molecular and Medical Biotechnology

Parameters for Prefix-Free Parsing: Analysis and Experimentation

Supervisors:
Zsuzsanna Lipták
Francesco Masillo

Candidate:
Martina Lucà
VR474960

Co-examiners:
Alejandro Giorgetti
Giovanni Malerba

Academic year: 2022-2023

"If there is such a thing as complete happiness, it is knowing that you are in the right place."

— Fannie Flagg, Fried Green Tomatoes at the Whistle Stop Cafe

Dedicated to all the people that pushed me forward and made a new place feel like home.

Contents

Abstract

Advances in DNA sequencing technologies have had a strong impact both in the biotechnology and bioinformatics fields, spanning from the early days of the Human Genome Project to contemporary initiatives like the 100 000 Genome Project and the Genome Trakr Network. A recurring theme is the exponential growth of data, with estimates projecting a need for 40 exabytes to store worldwide genome-sequence data by 2025. This emphasizes the critical importance of developing methods to efficiently store, handle, and analyze the vast amount of genomic data being generated.

The Burrows-Wheeler Transform (BWT) is a fundamental string transform that is found at the base of many bioinformatics tools (e.g., bwa and bowtie). A recently introduced heuristics based on Prefix-Free Parsing (PFP) is a very efficient algorithm to build the BWT on very large inputs, but its exact properties are poorly understood.

This research studies the significance of choosing appropriate parameters, namely modulus (m) and window size (w), of the PFP algorithm by systematically varying them to assess their impact on dictionary and parse size. The purpose is to better understand how the algorithm excels in handling big genomic data.

To achieve this, we used both real and artificial datasets, the latter generated by introducing controlled mutations with a certain probability to mimic genetic variation. By exploring mutation probabilities ranging from 0% to 10%, we aim to test the algorithm on strings with high to low sequence repetitiveness. The real datasets serve as a reference, allowing for comparison and validation of the algorithm's performance in real-life scenarios.

Methodologically, we tested all combinations of significant values of m and w on both real and artificial datasets. C++ and Python were used for algorithm implementation and to visualize the results of the tests.

1 Introduction

Advances in DNA sequencing technology had a large impact on both the biotechnology and bioinformatics fields. From the pioneering days of the Human Genome Project to the modern initiatives aiming to sequence thousands of genomes - e.g., the 100 000 Genome Project (Allard et al., 2016) and the GenomeTrakr Network (Turnbull et al., 2018) - a leitmotif can be found: the exponential growth of data generated. This incredible increase in data production is mainly due to the decrease in sequencing cost (see **Figure 1.1**).

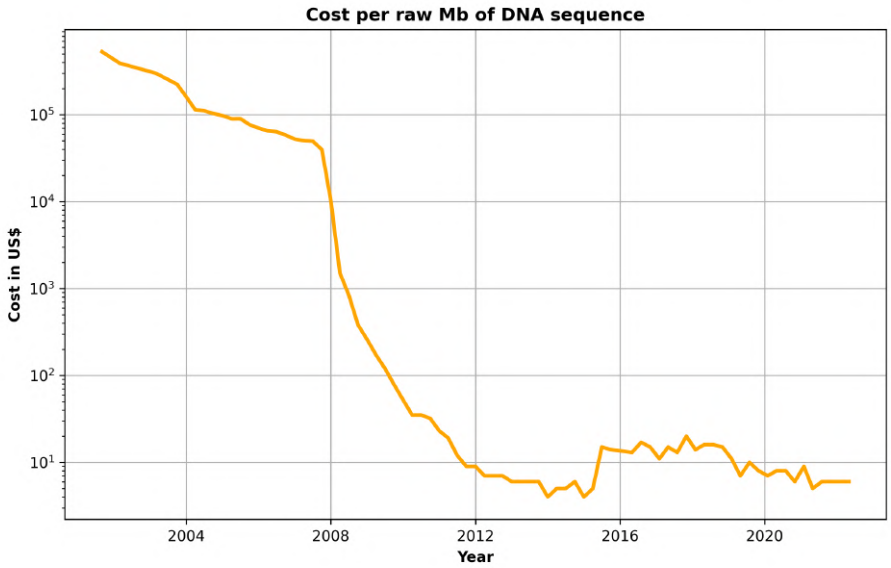


FIGURE 1.1 Plot illustrating the cost of DNA sequencing over time (measured in dollars per million bases). The logarithmic scale on the y-axis highlights the significant reduction in costs, with a notable acceleration starting in 2008, marking the transition to 'second-generation' DNA sequencing technologies. (Plot adapted from *DNA Sequencing Costs: Data* 2023)

To put things more in perspective, the National Human Genome Research Institute estimates that by 2025, we will need 40 exabytes to store the genome-sequence data generated

worldwide (*Genomic Data Science Fact Sheet* 2022). This underscores the importance of developing efficient methods to store, handle, and analyze this data.

1.1 Text indexes

A text index is a data structure built on the text that allows fast pattern matching without having to scan the text, and ideally in time independent of the size of the text. Pattern matching and other string processing tasks are often required in genomic studies, and as subroutines in sequencing algorithms (e.g. overlap finding / pairwise overlap finding, mapping to a reference, ...). Therefore, it is crucial to develop text indexes that are efficient, especially in view of the rapidly growing genomic data available.

1.1.1 Main indexes used in genomics

In the field of genomics, three main full-text indexes stand out: Suffix Trees (Weiner, 1973), Suffix Arrays (Gonnet et al., 1992; Manber and Myers, 1993) and the Burrows-Wheeler Transform, or BWT, (Burrows and Wheeler, 1994), which is at the heart of compressed text indexes such as the FM-index (Ferragina and Manzini, 2000).

A string $T = T[1..n]$ is a finite sequence of characters from an alphabet Σ. We use array-based notation for strings, and index strings from 1.[1] The i^{th} character is denoted $T[i]$ and $T[i..j]$ denotes the substring from the i^{th} to j^{th} character, for $1 \leq i \leq j \leq n$. The i^{th} suffix of T, denoted $suf_i(T)$, is the substring starting from position i, i.e., $suf_i(T) = T[i..n]$. The last character $T[n]$ of a string $T[1..n]$ is assumed to be an end-of-string character, denoted \$, where $\$ \notin \Sigma$. This character \$ does not occur elsewhere in the string, and it is strictly smaller than all characters $c \in \Sigma$.

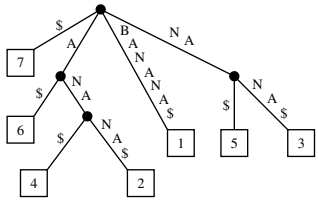
Suffix Trees. The Suffix Tree ST of a string T is the compact trie of the set of suffixes of T with edge labels from $(\Sigma \cup \$)^+$ such that:

- the labels of all edges outgoing from a node begin with different characters;

[1] In our codes, the indexing is from 0, due to the properties of the programming languages used. Therefore, we also use 0-based indexing in our pseudo-codes.

- the paths from the root to the leaves of ST spell the suffixes of S;
- each node in ST is either the root, a leaf, or a branching node.

Example : the ST of the string `BANANA$` is:



Suffix arrays. The suffix array (SA) of a string T is an array of integers listing the starting positions of the suffixes of T in lexicographical order. A suffix is a substring of the original string that starts at a particular position and extends to the end of the string. More formally:

Definition 1.1.1 (*Suffix array*)

The Suffix Array $SA[1..n]$ of a string T, with $|T| = n$, is a permutation of $[1..n] = \{1, 2, \ldots, n\}$ that satisfies the following condition:

$$\text{for } i = 1, \ldots, n-1 : \underbrace{T[SA[i]..n]}_{suf_{SA[i]}} <_{lex} \underbrace{T[SA[i+1]..n]}_{suf_{SA[i+1]}}$$

Example : The SA of the string `BANANA$` is:

$$SA = [7, 6, 4, 2, 1, 5, 3]$$

Despite their substantial space requirements, these indexes provide rapid and efficient pattern matching, which is crucial for many tasks in genome analysis. For example, consider the human genome, which is $3.2 \cdot 10^9$ bp long. Assuming an alphabet of size $\sigma = 4$, the suffix tree takes around 15 bytes per character and the suffix array takes around 4 bytes per character (Ganguly et al., 2020)[2]. Therefore, the suffix array of the human genome would take approximately 128 GB, while the suffix tree 480 GB. However, the genome itself can be represented in just $6.4 \cdot 10^9$ bits or approximately 0.8 GB.

BWT-based indexes are more space-efficient than both suffix arrays and suffix trees. They take $n \log \sigma(1+o(1))$ bits for a genome of length n on an alphabet of size σ (in uncompressed

[2]DNA and RNA are examples of alphabets of size 4: $\Sigma_{DNA} = \{A, C, G, T\}$; $\Sigma_{RNA} = \{A, C, G, U\}$

form) and support many types of queries that are necessary for high-throughput genome analysis approximately as fast as suffix arrays and suffix trees.

1.1.2 The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) of a string T is a permutation of T obtained by scanning its Suffix Array (SA) and taking the character preceding the one in each position of the Suffix Array (see Figure 1.2). More formally:

Definition 1.1.2 (*Burrows-Wheeler Transform*)

Let $T = t_1 t_2 \dots t_n$ be a string over Σ. The Burrows-Wheeler Transform of T, denoted by BWT is the permutation $L[1..n]$ of T defined as follows:

$$L[i] = \begin{cases} T[SA[i] - 1] & \text{if } SA[i] > 1 \\ \$ & \text{if } SA[i] = 1 \end{cases}$$

(Mäkinen et al., 2015)

```
n = length(T)

for (i < n){
    if (SA[i] != 0){
        BWT[i] = T[SA[i] - 1]
    }
    else{
        BWT[i] = T[n - 1]
    }
}
return BWT
```

LISTING 1 Pseudocode for the construction of the BWT from SA.

```
     1 2 3 4 5 6 7
T =  B A N A N A $

BWT = A N N B $ A A

SA = 7 6 4 2 1 5 3
     $ A A A B N N
       $ N N A N A
         A A N $ N
         $ N A   A
           A N   $
           $ A
             $
```

FIGURE 1.2 The BWT of string $T =$ BANANA$, shown for clarity with the suffix array of T and with the corresponding set of lexicographically sorted suffixes of T.

1.1.3 Prefix-Free Parsing for building big BWTs

Prefix-free Parsing (Boucher et al., 2019) was introduced for the construction of the Burrows-Wheeler Transform (BWT). It involves dividing the input text into overlapping variable-length phrases, which are then stored in a dictionary. The parse obtained from this dictionary can then be used to construct the BWT of the text efficiently. The key property of PFP is that the BWT of the text can be constructed from the dictionary and parse, using working space proportional to their total size and in $O(|T|)$ time.

The authors of the original paper state that the dictionary and parse generated from repetitive texts are significantly smaller than the original text, making it feasible to build compressed indexes for large genomic databases.

Dictionary construction

The main idea is to slide a window of a certain length (w) over the text and, whenever the Karp-Rabin (KR) hash (Karp and Rabin, 1987) of the window is $0 \bmod m$, a phrase is terminated, and the next one begins. The strings whose hash equals 0 are referred to as trigger strings. These phrases are then stored in a lexicographically sorted dictionary (D).

Lemma 1.1.1 (*Karp-Rabin Hash*)

The Karp-Rabin hash is a rolling hash function that calculates the hash value (i.e., a unique numerical value) of a string or its substrings. The hash value is computed using a chosen base (b) and modulus (p).

Example : Assume trigger strings: AT and GA

$T =$ ACGAAATTATTGATCTATTAATTAATACGA

Distinct phrases: ACGA, GAAAT, ATTAT, ATTGA, GATCTAT, ATTAAT, ATACGA

Dictionary $D =$ { 1:ACGA, 2:ATACGA 3:ATTAAT, 4:ATTAT, 5:ATTGA, 6:GAAAT, 7:GATCTAT }

Parse $P =$ 1,6,4,5,7,4,3,2

1.2 Goal of the thesis

This thesis aims to extensively study the impact of key parameters on the size of the dictionary and parse generated by the PFP algorithm, namely modulus (m) and window size (w), as

well as sequence repetitiveness. We understand how the algorithm functions, evaluating its performance under varying conditions and assessing its robustness with different levels of sequence repetitiveness.

The significance of this research lies in understanding the reasons behind the PFP algorithm's effectiveness for big data. By systematically varying m and w and conducting experiments on both real and artificially generated datasets, this study aims to better understand the algorithm's behavior across a spectrum of conditions.

Artificial datasets, generated through controlled mutation probabilities, serve the dual purpose of testing the algorithm under different conditions and providing a reference point for comparison with real-world genomic data. The consideration of mutation probabilities ranging from 0% to 10% and the selection of specific organisms *(SARS-CoV-2*, *Salmonella enterica*, and *Saccharomyces cerevisiae*) ensures a comprehensive evaluation, with an effort to simulate via artificial datasets the natural genetic variation observed in the chosen samples.

Methodologically, the research involves testing different m and w combinations of interest on real and artificial datasets. C++ (version 11.4.0) is employed to compute dictionary sizes, while Python (version 3.10.12) is used to run experiments in parallel and generate visualizations such as 3D plots and heatmaps for better presentation of the results. The comparison between real and artificial data establishes a baseline for algorithm performance while exploring extreme scenarios aids in studying algorithm behavior in worst-case conditions.

Anticipated outcomes include insights into the correlation between sequence repetitiveness and dictionary size, with the hypothesis that more repetitive sequences will result in smaller dictionaries. The study also aims to identify outliers and worst-case scenarios, providing insights into the algorithm's behavior.

This research contributes to the field of computational genomics by enhancing our understanding of the PFP algorithm, a relatively new and impactful tool in constructing the BWT of very large datasets.

2 Materials and methods

2.1 Experimental setup

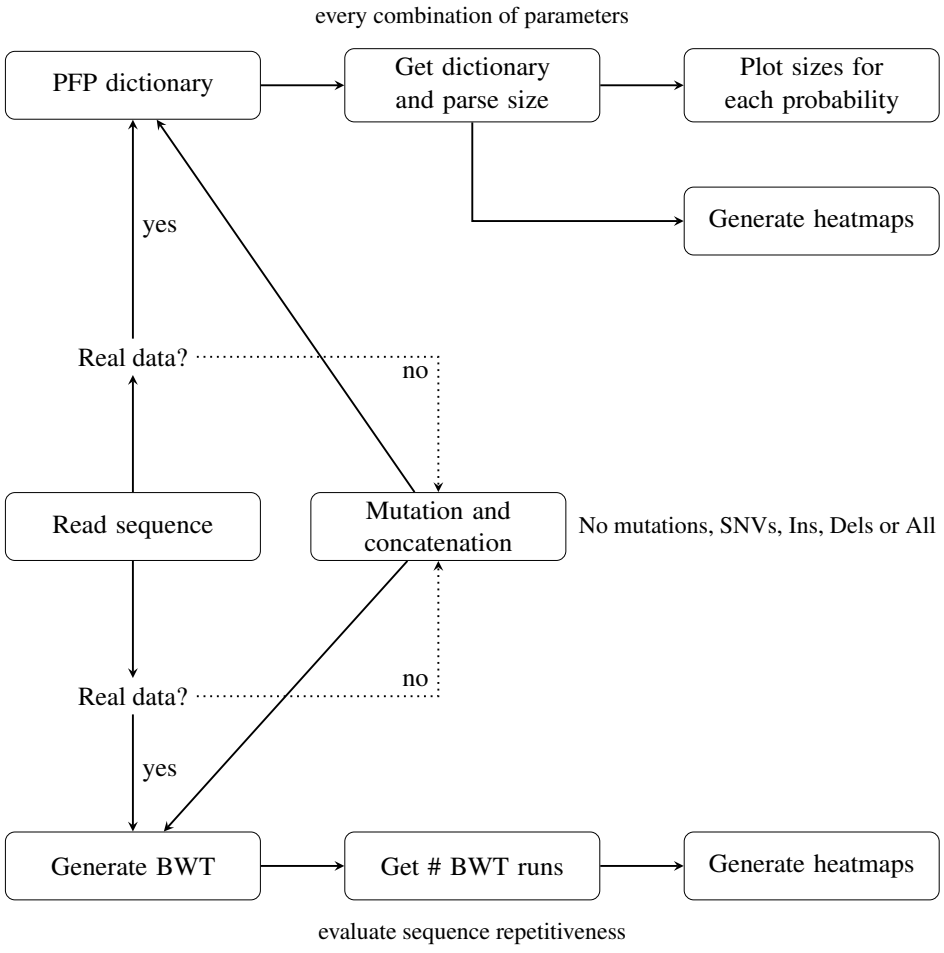


FIGURE 2.1 General scheme for the experimental setup.

The experimental setup in **Figure 2.1** was developed to evaluate how much the variation of parameters and the repetitiveness of input data affects dictionary and parse size. This work-

flow can be divided into two main processes: dictionary generation via PFP (implementation adapted from *giovmanz / Big-BWT · GitLab* 2024) and BWT generation (with *libsais* 2024).

Input data

In both the main processes, two different sets of data can be used:

- Real collections of genomes from *SARS-CoV-2*, *Saccharomyces cerevisiae* and *Salmonella enterica*.
- Artificial collections of genomes created by generating different variations of an original sequence and concatenating them (more details in the following sections).

2.1.1 Effect of the of parameters m and w

We tested all combinations of modulus (m) and window size (w) - where $m = [50, 100, 200, 400, 800]$ and $w = [6, 8, 10]$ as described in the tests performed by the authors of the original article - on both real data and artificial data (with different probabilities of change $P_{change} = [0, 0.0001, 0.001, 0.01, 0.1, 1.0]$). To do so, a Python (version 3.10.12) script was implemented, and the resulting dictionary and parse sizes for each combination of parameters were plotted via matplotlib (Hunter, 2007).

2.1.2 Effect of different kinds of mutations on dictionary size

To assess how much the kind of mutation affects the final dictionary size, we generated different matrices in the following fashion:

	$\mathcal{D}_{SNVs}$	$\mathcal{D}_{Ins}$	$\mathcal{D}_{Dels}$	$\mathcal{D}_{all}$
$\mathcal{D}_{SNVs}$	0	$\mathcal{D}_{SNVs} - \mathcal{D}_{Ins}$	$\mathcal{D}_{SNVs} - \mathcal{D}_{Dels}$	$\mathcal{D}_{SNVs} - \mathcal{D}_{All}$
$\mathcal{D}_{Ins}$	$\mathcal{D}_{Ins} - \mathcal{D}_{SNVs}$	0	$\mathcal{D}_{Ins} - \mathcal{D}_{Dels}$	$\mathcal{D}_{Ins} - \mathcal{D}_{All}$
$\mathcal{D}_{Dels}$	$\mathcal{D}_{Dels} - \mathcal{D}_{SNVs}$	$\mathcal{D}_{Dels} - \mathcal{D}_{Ins}$	0	$\mathcal{D}_{Dels} - \mathcal{D}_{All}$
$\mathcal{D}_{All}$	$\mathcal{D}_{All} - \mathcal{D}_{SNVs}$	$\mathcal{D}_{All} - \mathcal{D}_{Ins}$	$\mathcal{D}_{All} - \mathcal{D}_{Dels}$	0

To visualize the effect of different kinds of mutations, we generated two types of heatmaps with seaborn (Waskom, 2021):

- **Absolute value and single normalization.** For each combination of parameters, we take the absolute value of the difference matrix, and we apply normalization. This

kind of visualization helps us to identify which types of mutation generate the most different dictionary sizes.

- **Overall normalization.** We apply normalization over the whole set of combinations of parameters. This kind of visualization helps us to find outliers with exceptionally big or small dictionary sizes.

In particular, we used 0-1 normalization, also known as min-max normalization. This technique is used to rescale the dataset's values to a range between 0 and 1 to ensure that all the variables in the dataset have the same scale, preventing one variable from dominating the others. The formula for 0-1 normalization is given by:

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$

where:

- z_i is the normalized value
- x_i is the original value
- $\min(x)$ is the minimum value in the dataset
- $\max(x)$ is the maximum value in the dataset

2.1.3 Evaluation of sequence repetitiveness

We evaluate the sequence repetitiveness to understand better how much the mutation and concatenation process affects the final dictionary and parse size regardless of the parameter selection. We compute the BWT and count its runs (r) to do so. A run in the context of BWT refers to a sequence of identical consecutive characters in the transformed string.

Example : Input string: `BANANA$`, BWT: `BNN$AAA`, $r = 4$

Therefore, the BWT of a repetitive text is characterized by a relatively small number of rather long runs.

$$r \propto \frac{1}{\text{sequence repetitiveness}}$$

We generated another set of heatmaps to visualize how different kinds of mutations affect the degree of sequence repetitiveness.

2.2 Implementation

Our implementation of PFP is built around the need to study the changes in the dictionary and parse size with different kinds of strings. C++ (version 11.4.0) was used not only to compute the dictionary but also to introduce some changes in the reference sequence; in particular, we can generate:

- single nucleotide variants (SNVs);
- single nucleotide insertions;
- single nucleotide deletions;
- a mutation selected randomly among the three above.

This is done in random positions and with different probabilities of change. The modified sequences are then concatenated, and the resulting text is used as input for dictionary generation. The concatenation is done such that we can specify the final size we want in MB as argument, and the program will calculate how many times the original sequence needs to be repeated (N) to achieve that size ($maxsize$):

$$N = \frac{maxsize \cdot 10^6}{n}$$

where n is the size of the reference sequence.
These actions are implemented in different functions, which we will delve into in the following sections.

2.2.1 Dictionary generation

In our implementation (complete code can be found in **Listing 4**), the modulus and widow size are passed as arguments to facilitate the experiments, while the prime and base are fixed ($p = 1999999973$, $b = 256$).
Then, we can split the processing for the first window and all the successive ones:

- **First window.** The KR fingerprint of the first window is computed. If the substring is a trigger string, it is inserted in the dictionary. The KR fingerprint is calculated as follows:

$$KR_0 = \{[c_{w-1} \cdot b^0 + c_{w-2} \cdot b^1 + \cdots + c_0 \cdot b^{w-1}] \mod p\} \mod m \tag{2.1}$$

  where w is the window size, b is the base, p is the prime, m is the modulus and c is the character's ASCII value.[1]

- **Iterating over subsequent windows.** The hash values are updated, and we check if we have a trigger string. If that is the case, the substring between the previous and the current trigger strings is inserted in the dictionary; if it is already present, its counter is incremented. The KR fingerprint is calculated as follows:

$$KR_i = \{[(P_{i-1} - c_i \cdot b^{w-1}) \cdot b + c \cdot b^0] \mod p\} \mod m \tag{2.2}$$

  where P_{i-1} is the previous KR calculation before $\mod m$ and c_i is the rightmost character from the previous iteration.

Output. The dictionary is constructed using `std::unordered_map`, an associative container containing key-value pairs with unique keys. In our case, the key is a substring between trigger strings, and the value is the number of occurrences of said substring.
We compared our result with Manzini's original implementation output to double-check the dictionary size correctness.

2.2.2 Mutations generation

All the mutations are applied in a similar fashion, which follows this scheme:

- **Step 1.** The original sequence is copied and used as the base for mutation.

- **Step 2.** The number of positions to change (k) is determined based on the probability

[1] In the original paper p denotes the modulus, while m the prime.

of change (P_{change}) and the size of the reference sequence (n):

$$k = P_{change} \cdot n \tag{2.3}$$

- **Step 3.** The k positions to change are randomly drawn, and the mutations are induced with different methods.

SNVs generation

This function (complete code can be found in **Listing 5**) is used to create a mutated version of the reference DNA sequence by introducing substitutions of specific bases according to a given probability (see **Equation 2.3**).

We then use a fixed seed to generate random numbers in the $[0, n]$ range. Every time a new position is generated, it is memorized in a set (`std::set` [2]) until the set's size equals the number of extractions k. We then cycle over this set of positions in the reference, and we exchange the character in each position (c_i) with another character $\in \Sigma_3$, where $\Sigma_3 = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\} \setminus c_i$. If the input base is not one of the expected nucleotide bases, 'N' is returned to indicate an unknown base.

For optimization purposes, when $P_{change} = 1$, all the positions in the sequence are exchanged directly without using the set.

```
if(probability == 1){
    for (position in reference){
        reference[position]= RandomBase();
    }
}
else{
    while(positions_set < k){
        positions_set.insert(RandomPosition());
    }
    for (position in positions_set){
        temp_text[position] = RandomBase();
    }
}
```

LISTING 2 Pseudocode for the SNVs generation, where k is the number of mutations as described in **Equation 2.3** and n is the reference size.

[2] In C++ sets are containers that store unique elements following a specific order.

Insertions generation

This function (complete code can be found in **Listing 6**) modifies the reference DNA sequence by randomly inserting additional bases. It initializes a new sequence with the original DNA and introduces insertions based on a specified probability. This process involves using the `reserve()` function to preallocate memory for the temporary text in which we will perform the insertions; this is done to prevent unnecessary reallocations during the insertion process. By allocating enough space up front, the function minimizes the need for dynamic resizing, thereby improving efficiency.

Due to its poor performance in this context, we do not use the `insert()` function to perform the insertions themselves. Instead, we copy the text from position i to position j, where j is the position in which we want to insert a new base. Then, we randomly draw a base, add it to the copied text, and copy from position $j + 1$ to the next insertion's position.

```
while (positions_set < k){
    positions_set.insert(RandomPosition());
}
starting_position = 0;
for (position in positions_set){
    temp_text += reference[starting_position, position]
    temp_text += RandomBase();
    starting_position = position + 1;
}
temp_text += reference[starting_position, n];
```

LISTING 3 Pseudocode for the insertions generation, where k is the number of mutations as described in **Equation 2.3** and n is the reference size.

Deletions generation

This function modifies the reference sequence by randomly deleting bases (complete code can be found in **Listing 7**). To avoid conflicts, the removal is performed working from the end of the sequence.

3 | Results and discussion

3.1 PFP working space

In this section, we will analyze how the PFP algorithm's working space is affected by the choice of parameters (w and m) and the degree of sequence repetitiveness. This space is made up of two components:

- **Dictionary size** ($\mathcal{D}$) is affected mainly by modulus and degree of sequence repetitiveness. When patterns repeat frequently, we can represent them more efficiently by referencing previously seen substrings; this tends to result in smaller dictionary sizes.

- **Parse size** ($\mathcal{P}$) indicates the memory needed to represent parsing information. This size is tied to the length of phrases, and, unlike dictionary size, it is not affected by sequence repetitiveness. Parse size can be estimated as follows:

$$\log(\mathcal{D}) = \log\left(\frac{maxsize}{m}\right) \tag{3.1}$$

  where $\mathcal{D}$ is the dictionary size, $maxsize$ is the size of the collection and m is the modulus.

To better visualize these concepts, let us compare the dictionary and parse sizes obtained by running the algorithm on real and artificially generated datasets. In particular, we are going to consider a dataset generated by concatenating identical sequences until the $maxsize$ is reached (maximum repetitiveness).

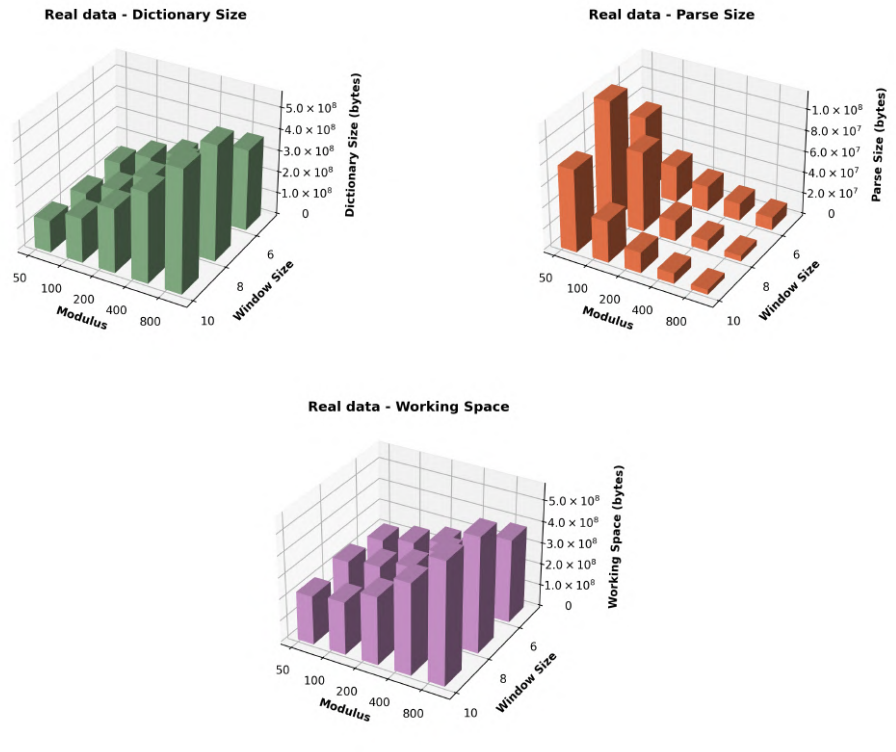


FIGURE 3.1 Dictionary, parse, and working space from PFP on a real *S. cerevisiae* dataset.

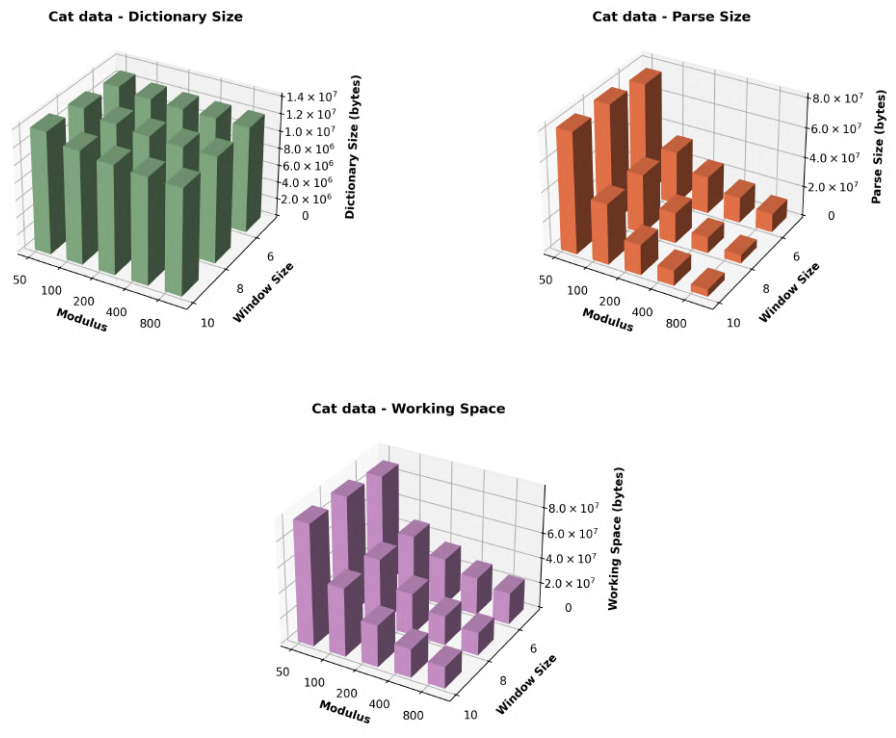


FIGURE 3.2 Dictionary, parse, and working space from PFP on the artificial *S. cerevisiae* dataset.

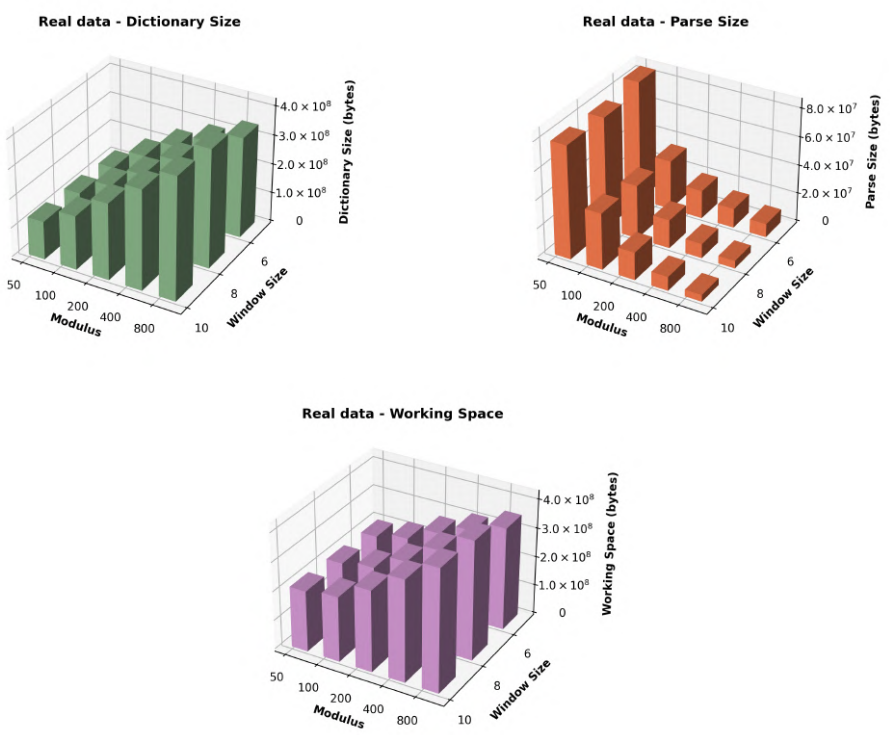


FIGURE 3.3 Dictionary, parse, and working space from PFP on a real *S. enterica* dataset.

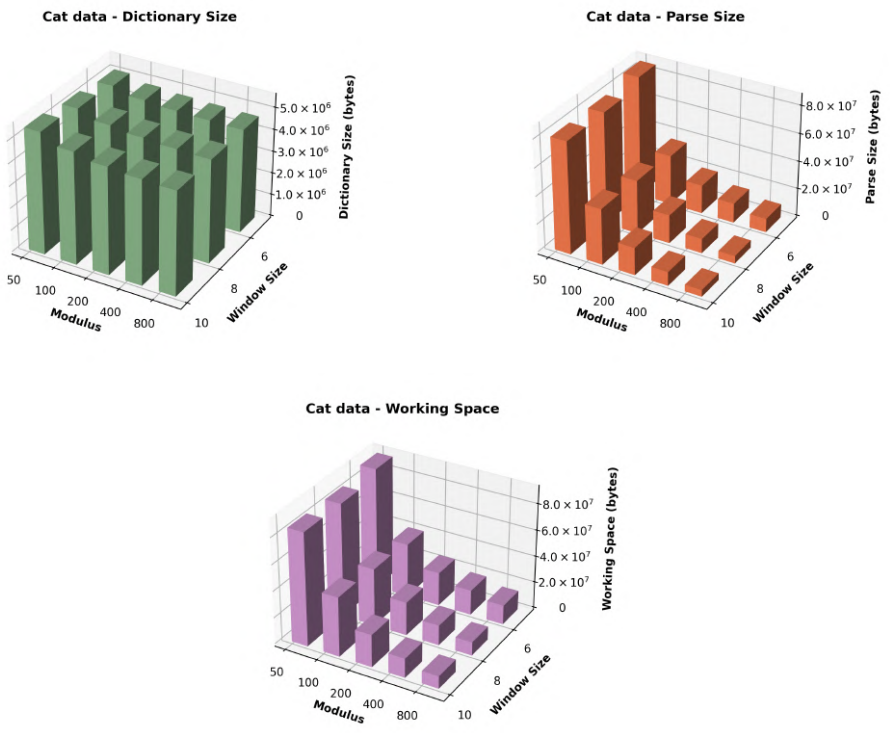


FIGURE 3.4 Dictionary, parse, and working space from PFP on the artificial *S. enterica* dataset.

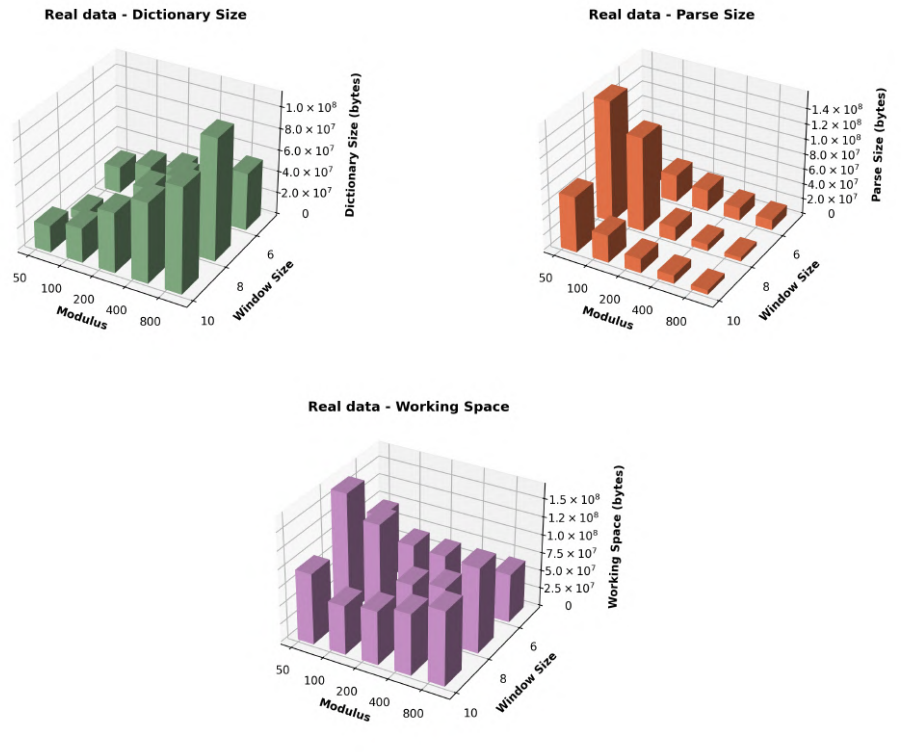


FIGURE 3.5 Dictionary, parse, and working space from PFP on a real *SARS-CoV-2* dataset.

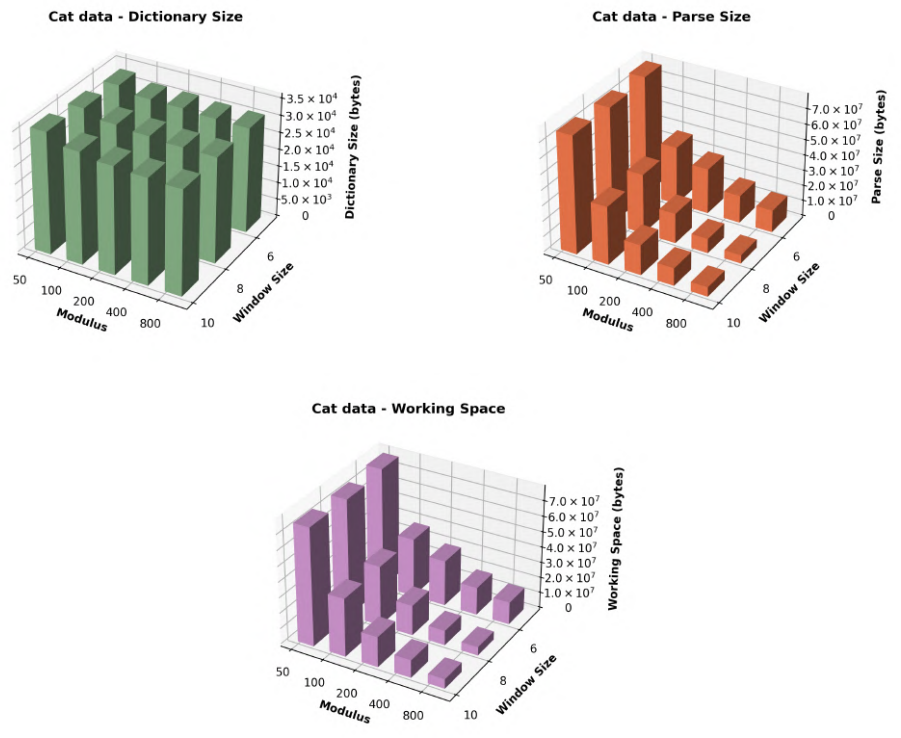


FIGURE 3.6 Dictionary, parse, and working space from PFP on the artificial *SARS-CoV-2* dataset.

Dictionary size. When looking at real data from both the *S. cerevisiae* and the *S. enterica* collections, we notice that $\mathcal{D}$ increases when m is increased, while w does not seem to have a significant impact. In contrast, we observe a different pattern when examining the artificial datasets. Despite changes in m or w, $\mathcal{D}$ remains relatively stable and is approximately equivalent to the size of a single sequence. This phenomenon underscores the influence of sequence repetitiveness on dictionary size.

We should note that with *SARS-CoV-2*, the results on the artificial dataset are consistent with the other two samples, while the ones from the real dataset are different in that they seem to be more influenced by w than the others. This can be due to the sequence size, which is much smaller than both *S. cerevisiae* and *S. enterica*.

Sample	Sequence length
SARS-CoV-2	30 kbp
S. enterica	4.6 Mbp
S. cerevisiae	12 Mbp

TABLE 3.1 Samples with their respective single sequence length.

As expected, sequence repetitiveness plays a crucial role because identical sequences result in just one entry in the dictionary, effectively reducing its size. Moreover, the modulus parameter directly affects dictionary size by determining the average phrase length: increasing m results in longer phrases, which in turn leads to an increase of $\mathcal{D}$.

Parse size. The situation changes when looking at $\mathcal{P}$: the parse size remains in the same order of magnitude in both the kinds of datasets and tends to decrease with increasing m values. This indicates that parse size is not influenced by sequence repetitiveness. This is expected since, in the parse, phrases are repeated just like in the text, irrespective of whether they are unique or not. Therefore, variations in sequence repetitiveness do not have an impact on parse size.

However, parse size is affected by the length of phrases: longer phrases result in fewer entries in the parse, as each phrase represents a larger portion of the input sequence. This is coherent with the observed decrease of parse size with increasing values of m.

3.2 The effect of sequence repetitiveness

3.2.1 Dictionary size

To better understand the effect of sequence repetitiveness, we have expanded our analysis by creating additional artificial datasets with three kinds of mutation (SNV, insertion, or deletion) applied at varying probabilities of change. When mutations with different probabilities are introduced into the dataset, it affects the overall repetitiveness of the sequence. Higher mutation probabilities lead to greater sequence diversity as more variations are introduced, resulting in decreased repetitiveness. Moreover, different kinds of mutation should affect sequence repetitiveness in different ways.

By comparing the performance of the PFP algorithm on datasets with varying mutation probabilities, we can observe how dictionary size responds to changes in sequence repetitiveness. Specifically, we expect higher probabilities to result in larger dictionaries due to increased sequence diversity.

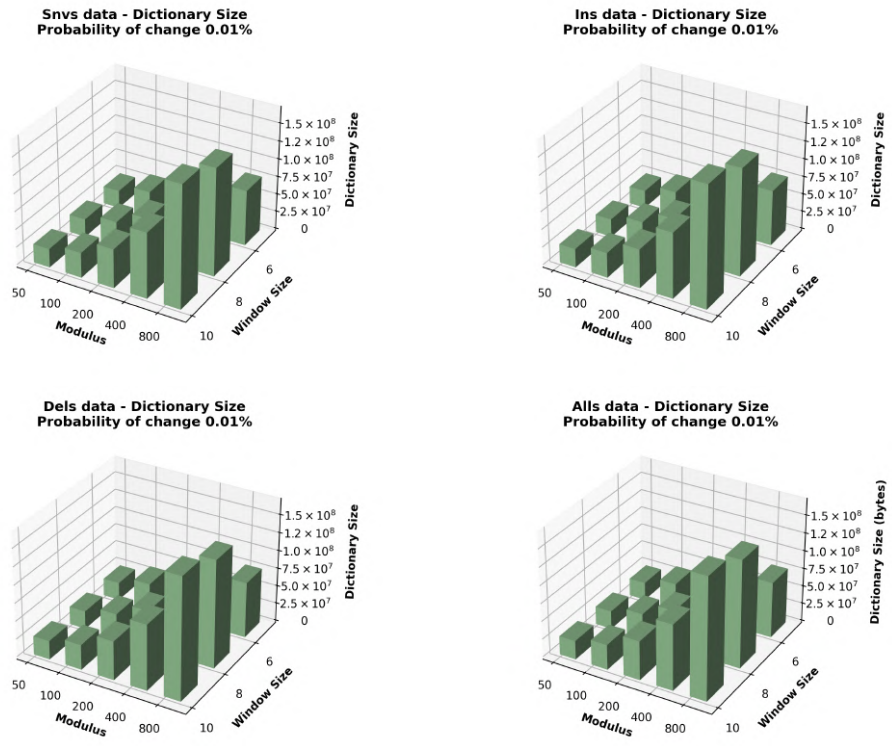


FIGURE 3.7 Dictionary size with different parameters, probability of change, and mutation kind for *S. cerevisiae*. (1/3)

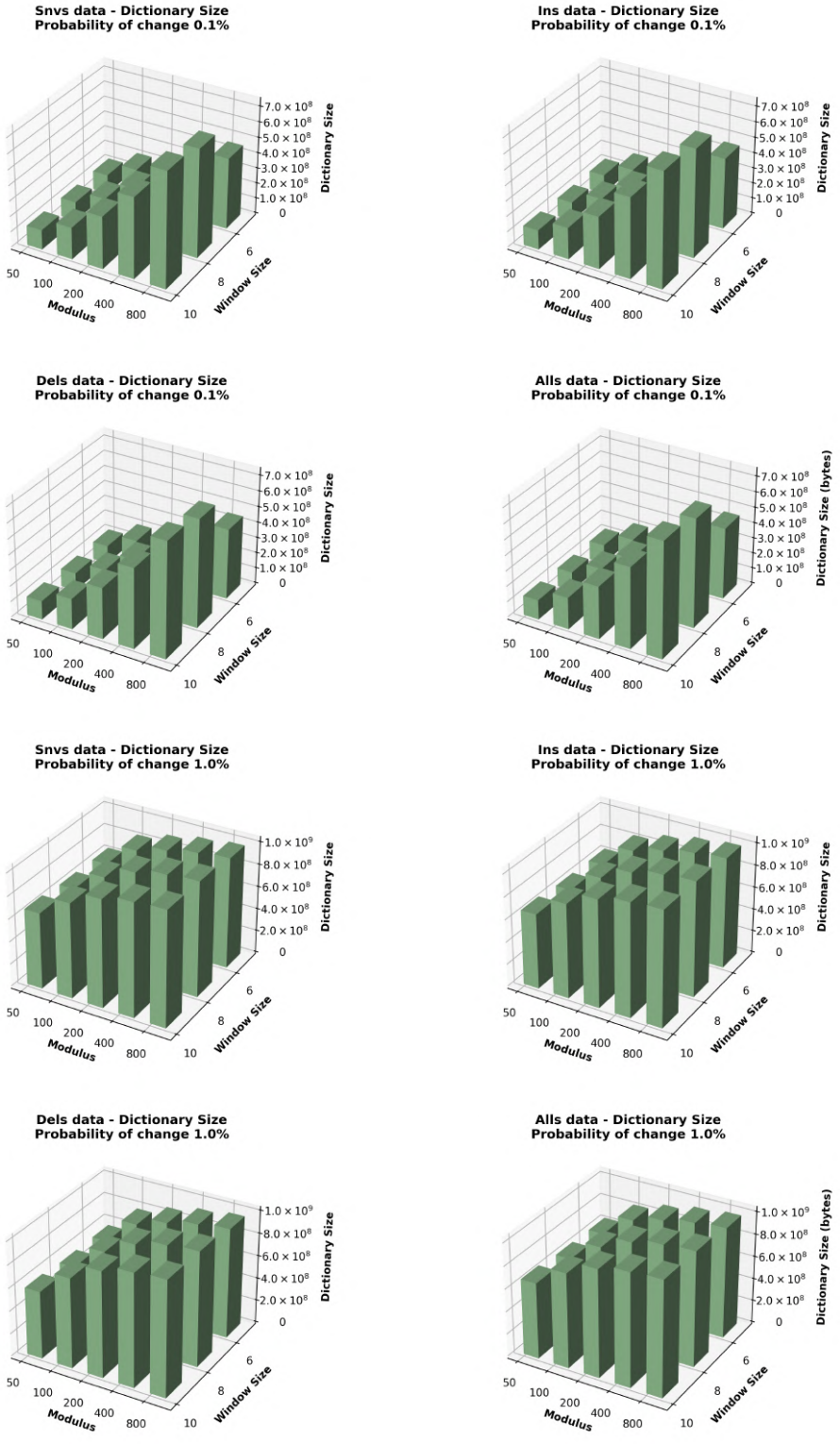


FIGURE 3.8 Dictionary size with different parameters, probability of change, and mutation kind for *S. cerevisiae*. (2/3)

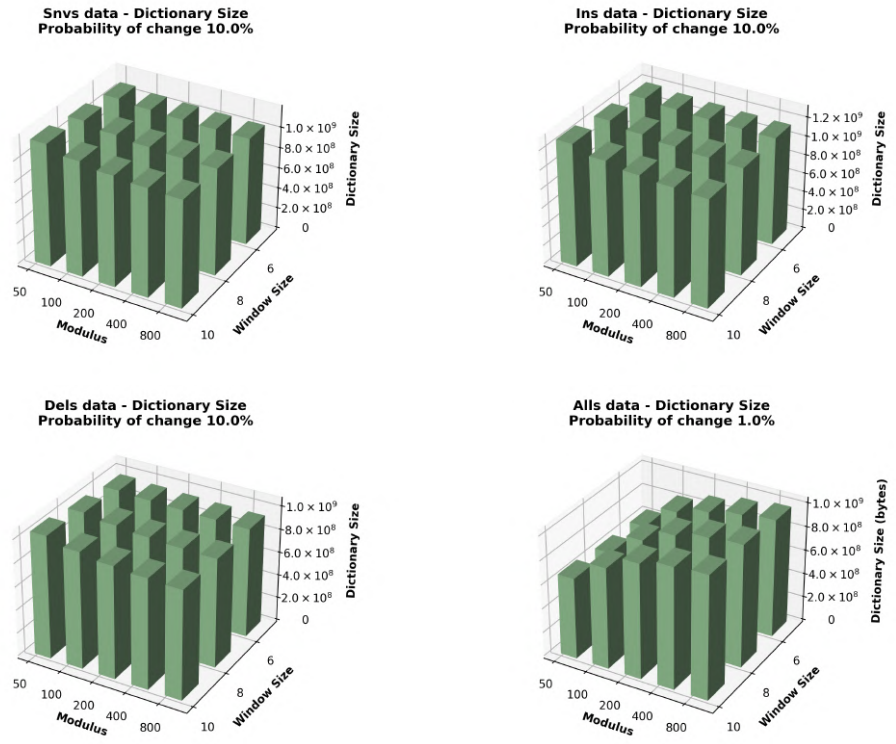


FIGURE 3.9 Dictionary size with different parameters, probability of change, and mutation kind for *S. cerevisiae*. (3/3)

Dictionary size in *S. cerevisiae* behaves as expected: the increase of sequence diversity results in an increase of $\mathcal{D}$. We do not have data regarding the within sequence identity of *S. cerevisiae*. However, we know the highest sequence identity of *S. cerevisiae* and *S. paradoxus* is greater than 95% (Muller and McCusker, 2011). Therefore, we can expect a within-sequence identity greater than that. Sure enough, the dictionary sizes closer to the one from the real dataset are the ones coming from the artificially generated datasets with a probability of change of 0.1%. This is coherent also with the degree of sequence repetitiveness, which is the most similar to the real one when $P_{change} = 0.1\%$ (see **Figure 3.15**).

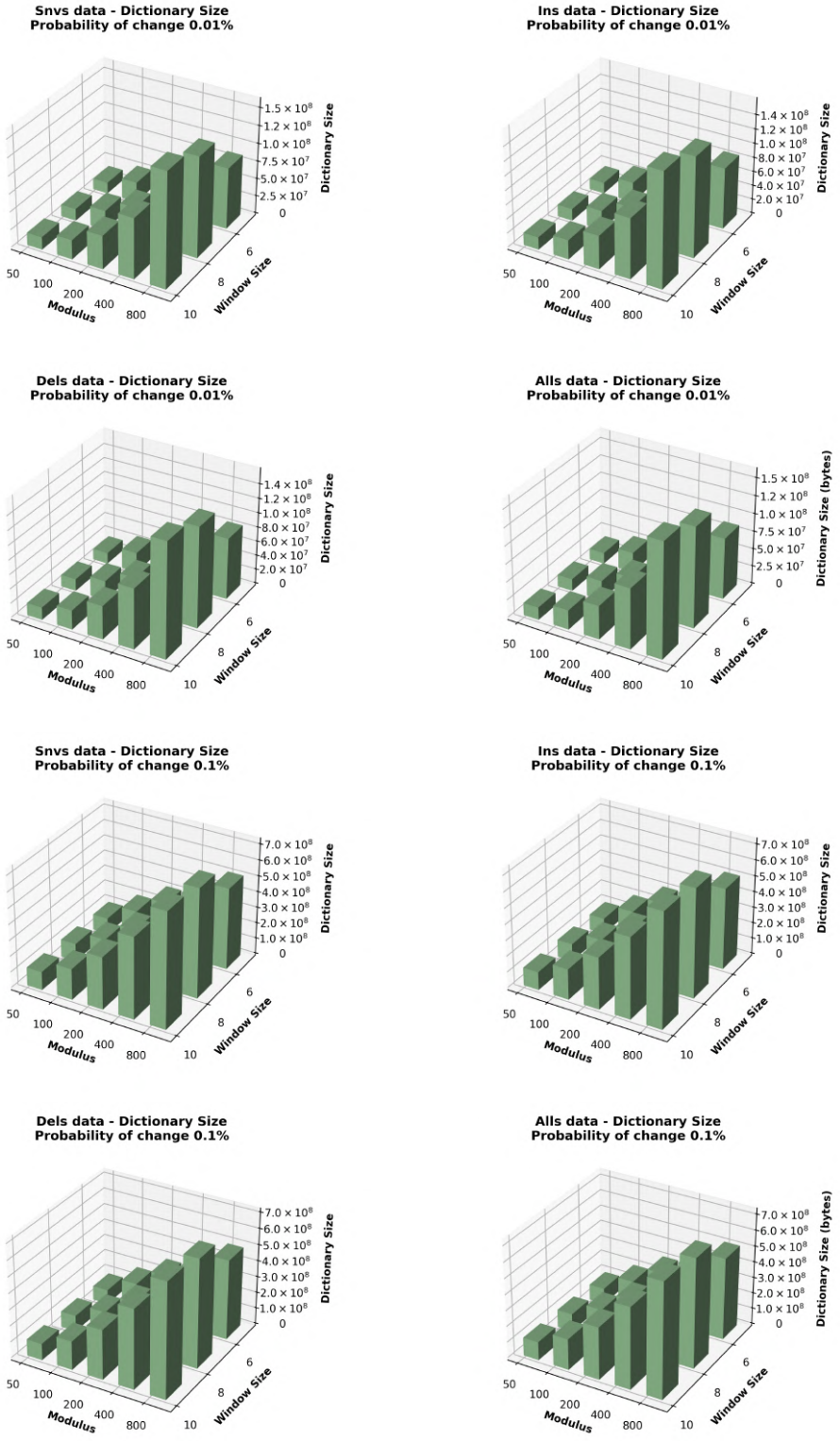


FIGURE 3.10 Dictionary size with different parameters, probability of change, and mutation kind for *S. enterica*. (1/2)

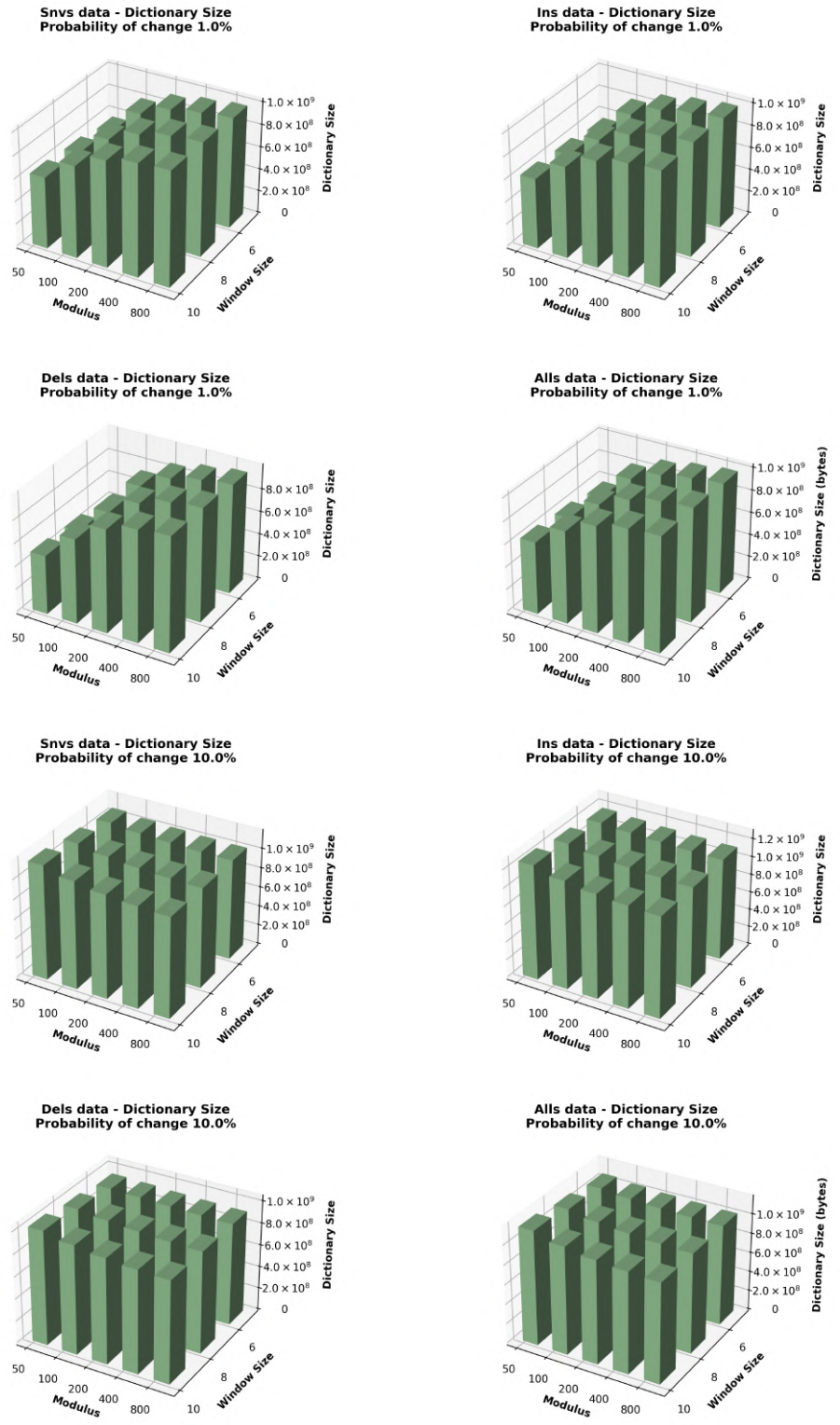


FIGURE 3.11 Dictionary size with different parameters, probability of change, and mutation kind for *S. enterica*. (2/2)

Dictionary size in *S. enterica* behaves as expected: the increase of sequence diversity results

in an increase of $\mathcal{D}$.

Sequence identity in *S. enterica* can vary significantly ranging from 98% to 99.99% of Average Nucleotide Identity (ANI) (Branchu et al., 2018). As expected, the results from the probability of change between 0.01% to 1% are most similar to those obtained from the real dataset. This is coherent also with the degree of sequence repetitiveness, which is the most similar to the real one when $P_{change} = 0.1\%$ (see **Figure 3.15**).

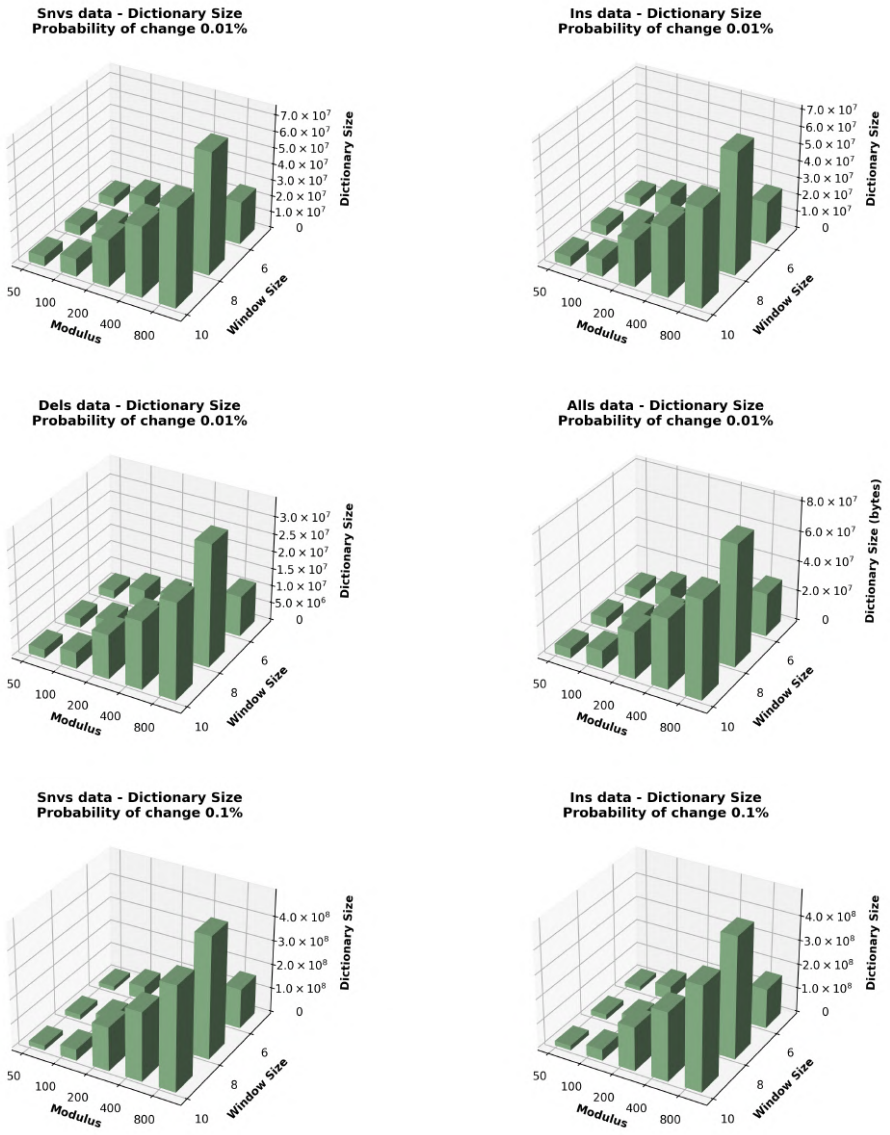


FIGURE 3.12 Dictionary size with different parameters, probability of change, and mutation kind for *SARS-CoV-2*. (1/3)

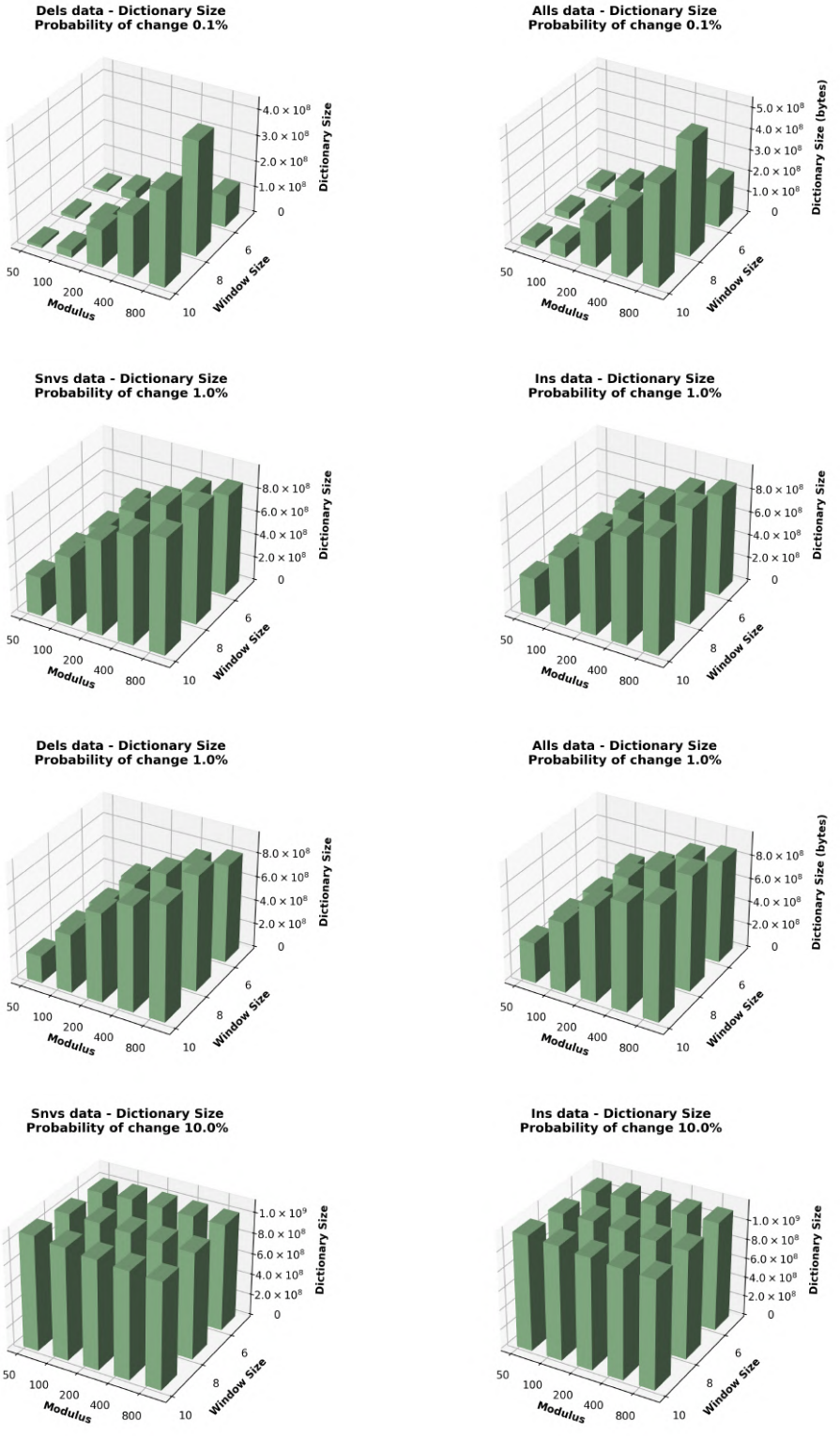


FIGURE 3.13 Dictionary size with different parameters, probability of change, and mutation kind for *SARS-CoV-2*. (2/3)

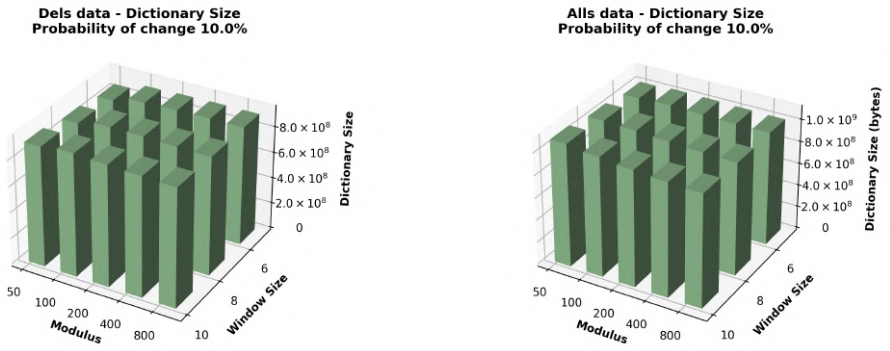


FIGURE 3.14 Dictionary size with different parameters, probability of change, and mutation kind for *SARS-CoV-2*. (3/3)

Dictionary size in *SARS-CoV-2* behaves as expected: the increase of sequence diversity results in an increase of $\mathcal{D}$.

Human *SARS-CoV-2* genomes have a sequence identity of 99.98% (Gómez-Carballa et al., 2020). However, the results with a probability of change of 0.1% are the most similar to the ones obtained with the real dataset. This is coherent with the degree of sequence repetitiveness, which is the most similar to the real one when $P_{change} = 0.01\%$ (see **Figure 3.15**).

3.2.2 Different kinds of mutations and dictionary size

In **Chapter B** of the appendix, all the heatmaps generated with two methods described in **Section 2.1.2** can be found.

The heatmaps generated with absolute value and single normalization show a similar trend:

- With $P_{change} = 10\%$ Dels and Ins are the most different, while All and SNVs are the most similar.
- With smaller P_{change}, the biggest differences are between Dels and all other kinds of mutations.

However, it should be noted that some exceptions with certain combinations of parameters within the same probability of change can be observed. Moreover, a notable exception can be observed in *SARS-CoV-2* with $P_{change} = 0.1\%$: in this case, the most significant difference

is between All and Dels, with the difference between Dels and SNVs, and Dels and Ins closely following.

Looking at the second kind of heatmaps, we can identify which type of mutation and combination of parameters gives the worst result (i.e., the most extensive dictionary size). When $P_{change} = 10\%$ the results from every sample type are the same: dictionary sizes obtained with every combination of parameters are comparable; this is coherent with what can be observed in the plots in **Figures 3.9**, **3.11**, **3.13**, **3.14**. Moreover, we can observe that Ins always results in the biggest dictionary size, while Dels in the smallest. However, the results change with different samples when P_{change} decreases:

- ***S. cerevisiae* and *S. enterica*.** When $P_{change} = 0.01\%$ the combinations of parameters $w = [8, 10]$ and $m = 800$ with the insertions, the SNVs, and the combination of the three kinds of mutations are the worst; this is coherent with the statement from **Section 3.1** ($\mathcal{D} \propto m$). When $P_{change} = 0.1\%$, the combination of parameters does not seem to affect dictionary size. When $P_{change} = 1\%$ the combinations of parameters $w = [6, 8, 10]$ and $m = 50$ with the insertions, the SNVs and the combination of the three kinds of mutations are the worst; this goes against $\mathcal{D} \propto m$.

- ***SARS-CoV-2*.** With $P_{change} = 0.01\%$, the combinations of parameters $w = [8, 10]$ and $m = 800$ with the insertions, the SNVs, and the combination of the three kinds of mutations are the worst; this is coherent with $\mathcal{D} \propto m$. The previous statement is also true with $P_{change} = 0.1\%$; additionally, the combinations of parameters $w = [10, 8]$ and $m = 400$ also show some of the worst results. With $P_{change} = 1\%$, the combinations of parameters $w = [6, 8, 10]$ and $m = [50, 100]$ with the insertions, the SNVs, and the combination of the three kinds of mutations are the worst; this goes against $\mathcal{D} \propto m$.

3.2.3 Different kinds of mutations and sequence repetitiveness

In the previous section, we established that SNVs and insertions result in the biggest dictionary sizes: theoretically, these two types of mutation should disrupt sequence repetitiveness more. To check this, we can look at the BWT runs we obtain from each kind of mutation at

every P_{change}.

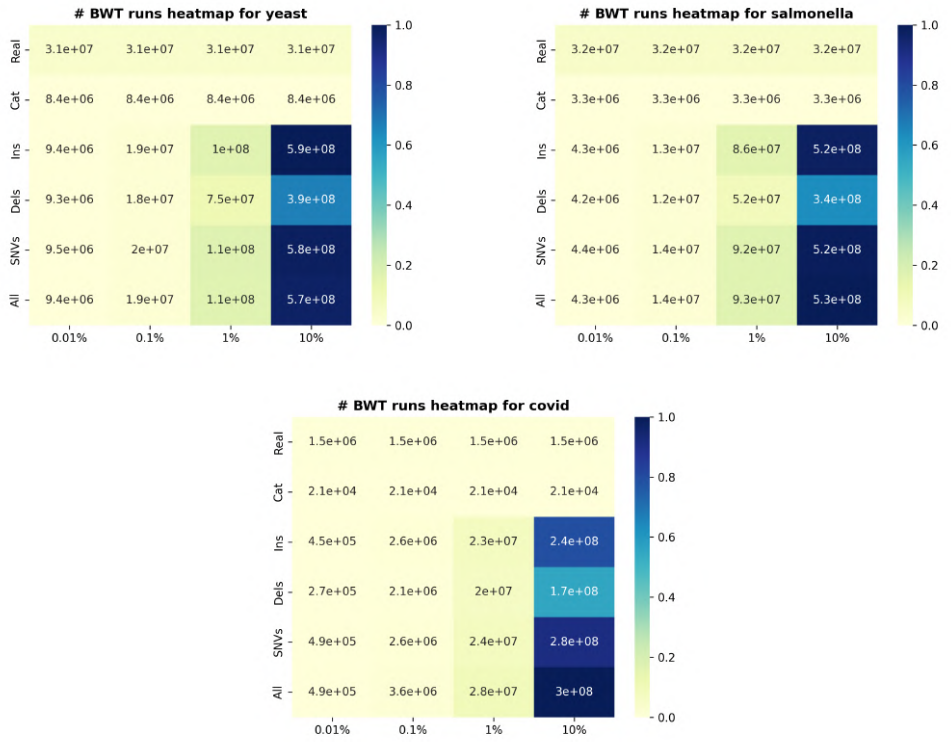


FIGURE 3.15 Heatmap representing the number of BWT runs for each kind of mutation at different P_{change} (r for the real and completely repetitive datasets is shown as reference).

For every sample type, we can observe that with increasing P_{change}, the number of BWT runs increases. Moreover, r is consistently higher with mutation types SNVs, Ins, and All. It should be noted that the sequence repetitiveness with the deletions is also less because we are shortening the sequence. However, the difference is remarkable and cannot be ascribed to the sole shortening of the sequence. Nevertheless, future experiments could address this by concatenating other mutated sequences to keep $maxsize$ constant.

4 Conclusion

In this thesis, we explored how the Prefix-Free Parsing (PFP) algorithm behaves, focusing on the experimentation of different combinations of parameters like modulus (m) and window size (w), and on the effect sequence repetitiveness. Our goal was to understand how these factors affect the dictionary size and parse (working space) produced by the algorithm.

Through our analysis of real and artificial datasets of organisms like SARS-CoV-2, *Salmonella enterica*, and *Saccharomyces cerevisiae*, we aimed to test the algorithm on a variety of strings with different degrees of sequence repetitiveness.
Our approach involved testing all combinations of significant values of m and w on a range of mutation probabilities, using C++ and Python for implementation and visualization of the results.

We found that the size of the dictionary primarily depends on modulus and sequence repetitiveness, while parse size is influenced mostly by m and remains relatively consistent across datasets with diverse degrees of sequence repetitiveness. Moreover, our analysis of mutation types revealed that deletions tend to increase sequence repetitiveness less than SNVs, insertions, and the combination of these three mutation types.

We believe that this study will lead to a deeper comprehension of the PFP algorithm and will help us understand its implications for genomic data analysis. Future work will be required to study PFP further, how it takes advantage of sequence repetitiveness, and to better understand which combinations of parameters work best under which circumstances.

A | Code

A.1 PFP implementation

```
std::unordered_map<std::string, uint64_t>
 ↪  GenerateDictionary(std::string &c_text, int windowsize,
 ↪  int modulus){
    uint64_t base = 256; // base (ASCII)
    int64_t prime = 1999999973; // first modulus for the
     ↪  parse (needs to be prime)
    std::string currentstring; // save here the chars and
     ↪  then save to dictionary when new trigger string is
     ↪  found
    std::vector<uint64_t> addends; // addends array for
     ↪  later karprabinhash
    std::unordered_map<std::string, uint64_t> dictionary; //
     ↪  dictionary

    // init current position = end of current window
    int currentpos = windowsize - 1;

    // FIRST WINDOW
    for (int i = 0; i < windowsize; i++){
        addends.push_back((int)c_text[currentpos - i] *
         ↪  LinearTimePower(base, i, prime));
```

```
}
uint64_t currenthash = 0;
for (int i = 0; i < addends.size(); i++){
    currenthash += addends[i];
    currenthash = currenthash % prime;
}
uint64_t karprabin = currenthash % modulus;
currentstring = c_text.substr(0, windowsize); // new
 ↪  current string
if (karprabin == 0){ // insert in dictionary
    auto insertion = dictionary.insert({currentstring,
     ↪  1});
}

// ALL SUCCESSIVE WINDOWS
uint64_t maxpower = LinearTimePower(base, (windowsize -
 ↪  1), prime);
for (int i = 0; i < c_text.size() - 1; i++){
    currentpos++;
    uint64_t nexthash = currenthash + (prime -
     ↪  ((int)c_text[currentpos - windowsize] *
     ↪  maxpower) % prime);
    nexthash = (base * nexthash +
     ↪  (int)c_text[currentpos]) % prime;
    karprabin = nexthash % modulus;
    currentstring.push_back(c_text[currentpos]);
    if (karprabin == 0){ // insert in dictionary
        auto insertion =
         ↪  dictionary.insert({currentstring, 1});
        if(insertion.second == false)
         ↪  insertion.first->second++;
```

```
            currentstring = c_text.substr(i + 1,
             ↪  windowsize); // new current string
        }

        currenthash = nexthash;
    }

    currentstring.append(windowsize, (char)0);
    auto insertion = dictionary.insert({currentstring, 1});

    return dictionary;
}
```

LISTING 4 Code for dictionary generation.

```
std::string GenerateSNVs (std::string &reference, uint
 ↪  maxsize, float probability) {
    // concatenated text string, with first sequence being
     ↪  the original one
    std::string c_text;
    c_text = reference;
    c_text += 1;

    // decide how many positions to change wrt probability
     ↪  of SNV
    int n_extractions = std::min((u_int64_t)(probability *
     ↪  reference.size()), reference.size());
    std::cout << "Number of extractions: " << n_extractions
     ↪  << '\n';

    // copy of reference to use to apply SNVs
    std::string temp_text;
```

```
temp_text += reference;

// generate random SNVs and concatenate the resulting
↪ text
auto rng = std::default_random_engine {};
std::cout << "Going to repeat SNVs: " << (maxsize *
↪ 1e6)/reference.size() << '\n';
std::mt19937 gen(0); // seed generator
std::uniform_int_distribution<> distr(0,
↪ reference.size()); // define the range
std::set<int> k_positions;
for(int i = 0; i < (maxsize * 1e6)/reference.size();
↪ i++){
    // do exactly k substitutions:
    if(probability==1){
        for(int i = 0; i < n_extractions; i++){
            temp_text[i] = GetOtherBase(temp_text[i]);
        }
    }
    else{
        while(k_positions.size() < n_extractions){
            k_positions.insert(distr(gen));
        }
        for (auto position: k_positions){
            temp_text[position] =
            ↪ GetOtherBase(temp_text[position]);
        }
        k_positions.clear();
    }
    // concatenate the changed text
    c_text += temp_text;
```

```
        c_text += 1;

        // regenerate original string to apply new SNVs
        temp_text = reference;
    }
    return c_text; // return the concatenation of mutated
    ↪ text
}
```

LISTING 5 Code for SNVs generation.

```
std::string GenerateInsertion (std::string &reference, uint
↪ maxsize, float probability){
    // concatenated text string, with first sequence being
    ↪ the original one
    std::string c_text;
    c_text = reference;
    c_text += 1;

    // decide how many positions to change wrt probability
    ↪ of change
    int n_extractions = std::min((u_int64_t)(probability *
    ↪ reference.size()), reference.size());
    std::cout << "Number of extractions: " << n_extractions
    ↪ << '\n';

    // copy of reference to use to apply changes
    std::string temp_text;
    // temp_text += reference;
    temp_text.reserve(reference.size() + n_extractions);
    // generate random insertions and concatenate the
    ↪ resulting text
```

```
auto rng = std::default_random_engine {};
std::cout << "Going to repeat insertions: " << (maxsize
↪  * 1e6)/reference.size() << '\n';
std::mt19937 gen(0); // seed generator
std::uniform_int_distribution<> distr(0,
↪  reference.size()); // define the range
std::set<int> k_positions;
for(int i = 0; i < (maxsize * 1e6)/reference.size();
↪  i++){
    while(k_positions.size() < n_extractions){ //
    ↪  generate exactly k insertions
        k_positions.insert(distr(gen));
    }
    int starting_position = 0;
    for(std::set<int>::iterator it =
    ↪  k_positions.begin(); it != k_positions.end();
    ↪  it++){
        // std::cout << *it << '\n';
        temp_text += reference.substr(starting_position,
        ↪  *it - starting_position);
        temp_text += GetABase();
        starting_position = *it;
    }
    if(starting_position < reference.size())
        temp_text += reference.substr(starting_position,
        ↪  reference.size() - starting_position);
    // assert(temp_text.size() == reference.size() +
    ↪  n_extractions);
    k_positions.clear();

    // concatenate the changed text
```

```
            c_text += temp_text;
            c_text += 1;

            // regenerate original string to apply new changes
            temp_text.erase();
            temp_text.reserve(reference.size() + n_extractions);
        }
        return c_text; // return the concatenation of mutated
        ↪ text
}
```

LISTING 6 Code for Insertions generation.

```
std::string GenerateDeletion (std::string &reference, uint
↪ maxsize, float probability){
    std::string c_text;
    c_text = reference;
    c_text += 1;

    // decide how many positions to change wrt probability
    ↪ of change
    int n_extractions = std::min((u_int64_t)(probability *
    ↪ reference.size()), reference.size());
    std::cout << "Number of extractions: " << n_extractions
    ↪ << '\n';

    // copy of reference to use to apply changes
    std::string temp_text;
    temp_text.reserve(reference.size());
    // temp_text += reference;
```

```
// generate random deletions and concatenate the
 ↪  resulting text
if(n_extractions < reference.size()){
    auto rng = std::default_random_engine {};
    std::cout << "Going to repeat deletions: " <<
     ↪  (maxsize * 1e6)/reference.size() << '\n';
    std::mt19937 gen(0); // seed generator
    std::uniform_int_distribution<> distr(0,
     ↪  reference.size());
    std::set<int> k_positions;
    for(int i = 0; i < (maxsize * 1e6)/reference.size();
     ↪  i++){
        temp_text.erase();
        temp_text.reserve(reference.size());
        while(k_positions.size() < n_extractions){
            k_positions.insert(distr(gen));
        }
        int starting_position = 0;
        for(std::set<int>::iterator it =
         ↪  k_positions.begin(); it !=
         ↪  k_positions.end(); it++){
            temp_text +=
             ↪  reference.substr(starting_position, *it -
             ↪  starting_position);
            starting_position = *it + 1;
        }
        if(starting_position < reference.size())
            temp_text +=
             ↪  reference.substr(starting_position,
             ↪  reference.size() - starting_position);
        k_positions.clear();
```

```
            // concatenate the changed text
            c_text += temp_text;
            c_text += 1;
        }
    }
    else{
        std::cout << "Number of extractions exceeds
        ↪ reference length!" << std::endl;
    }
    return c_text; // return the concatenation of mutated
    ↪ text
}
```

LISTING 7 Code for Deletions generation.

B | Heatmaps: dictionary size

B.1 *S. cerevisiae*

FIGURE B.1 Effect of probability of change 0.01% on dictionary size - Absolute value and single normalization

FIGURE B.2 Effect of probability of change 0.1% on dictionary size - Absolute value and single normalization

FIGURE B.3 Effect of probability of change 1% on dictionary size - Absolute value and single normalization

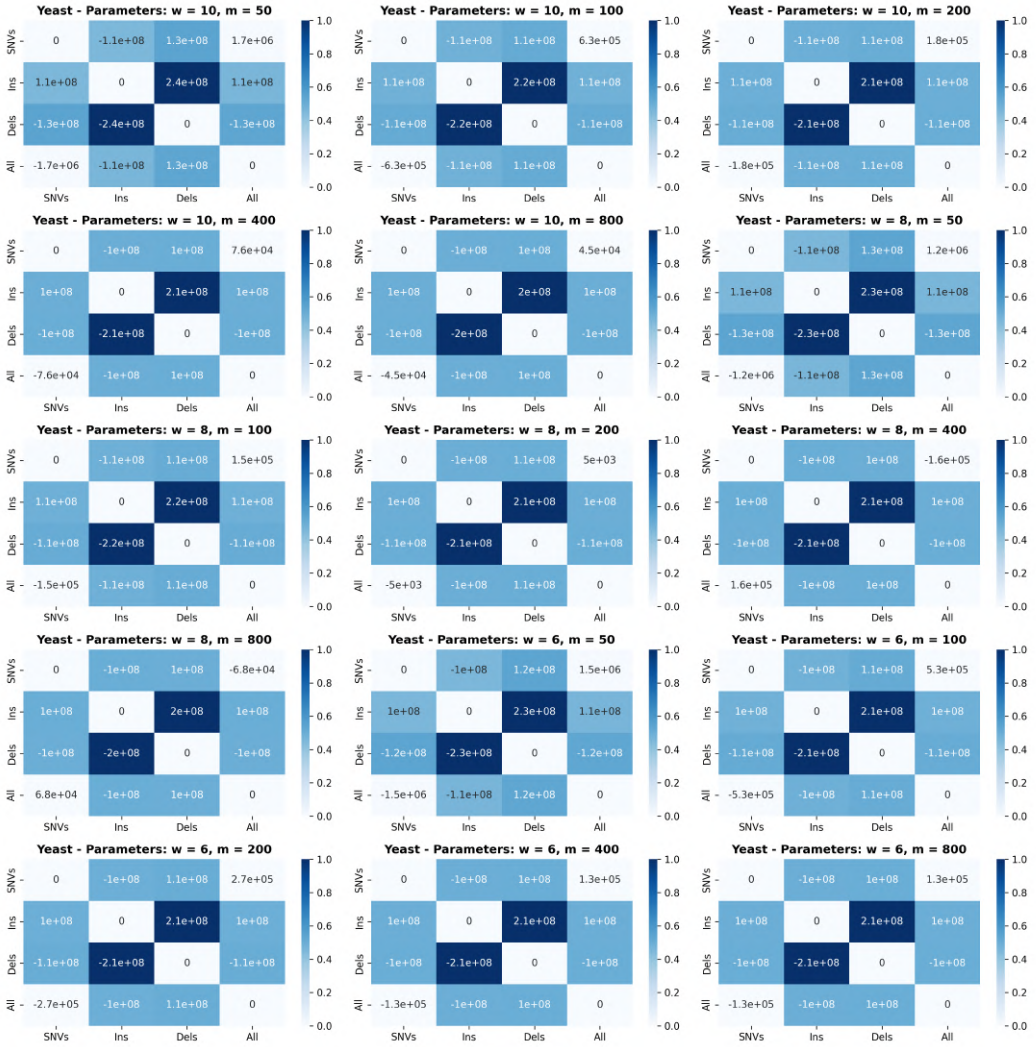


FIGURE B.4 Effect of probability of change 10% on dictionary size - Absolute value and single normalization

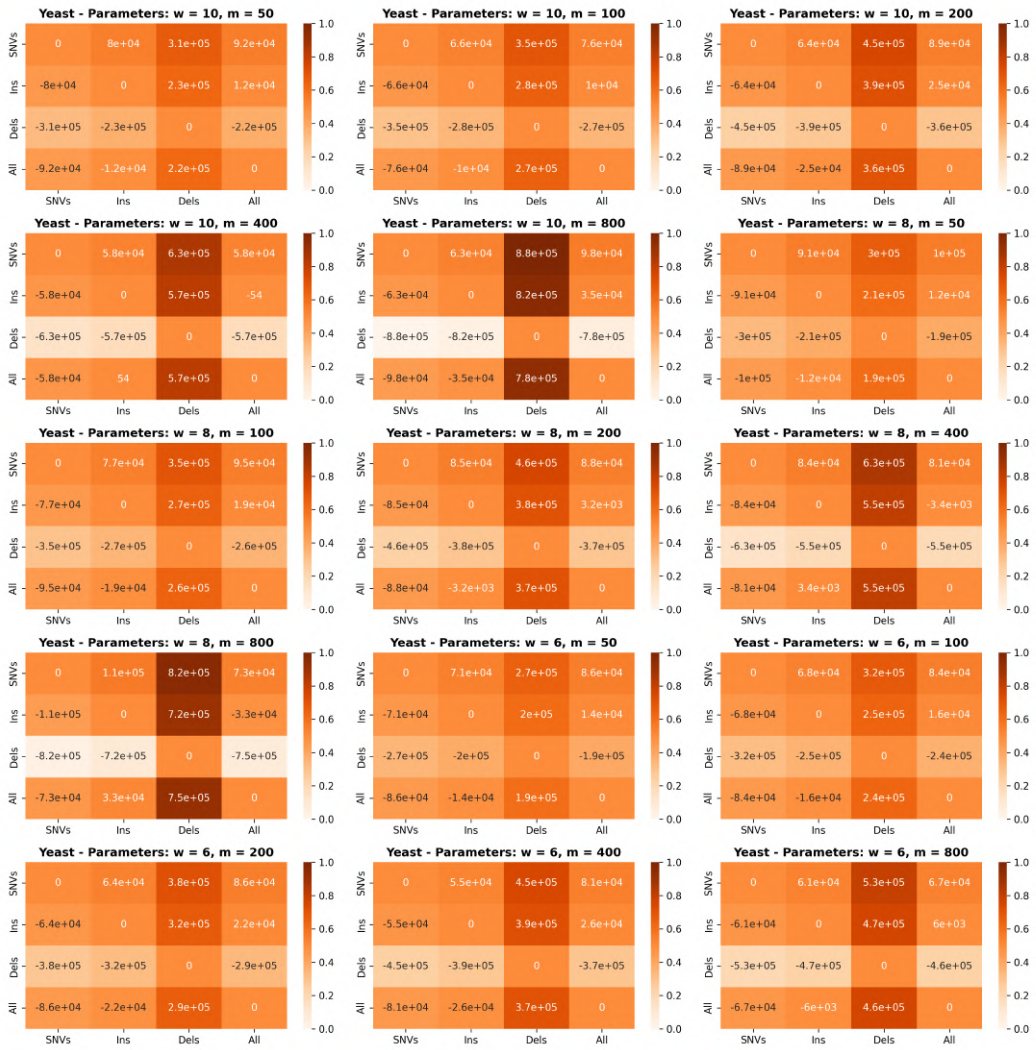


FIGURE B.5 Effect of probability of change 0.01% on dictionary size - Overall normalization

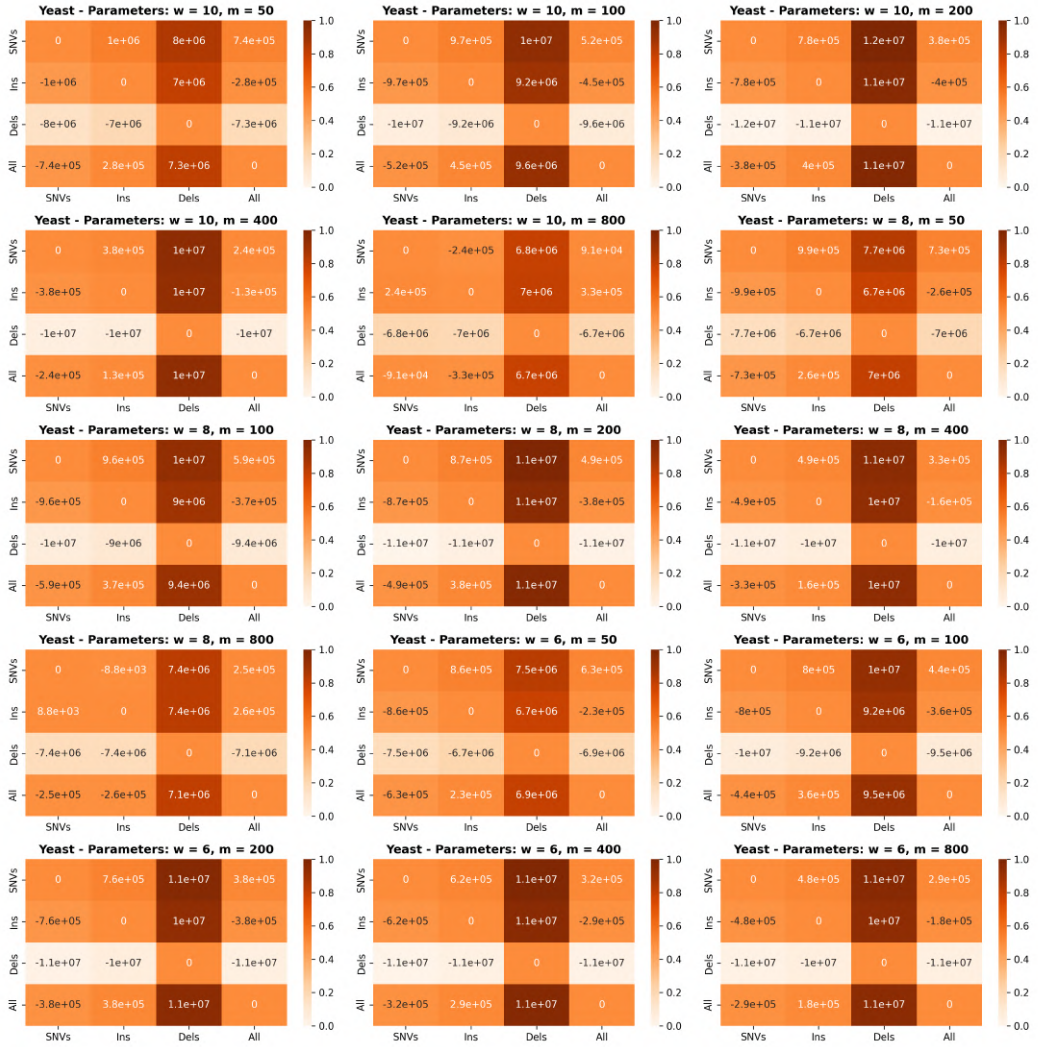


FIGURE B.6 Effect of probability of change 0.1% on dictionary size - Overall normalization

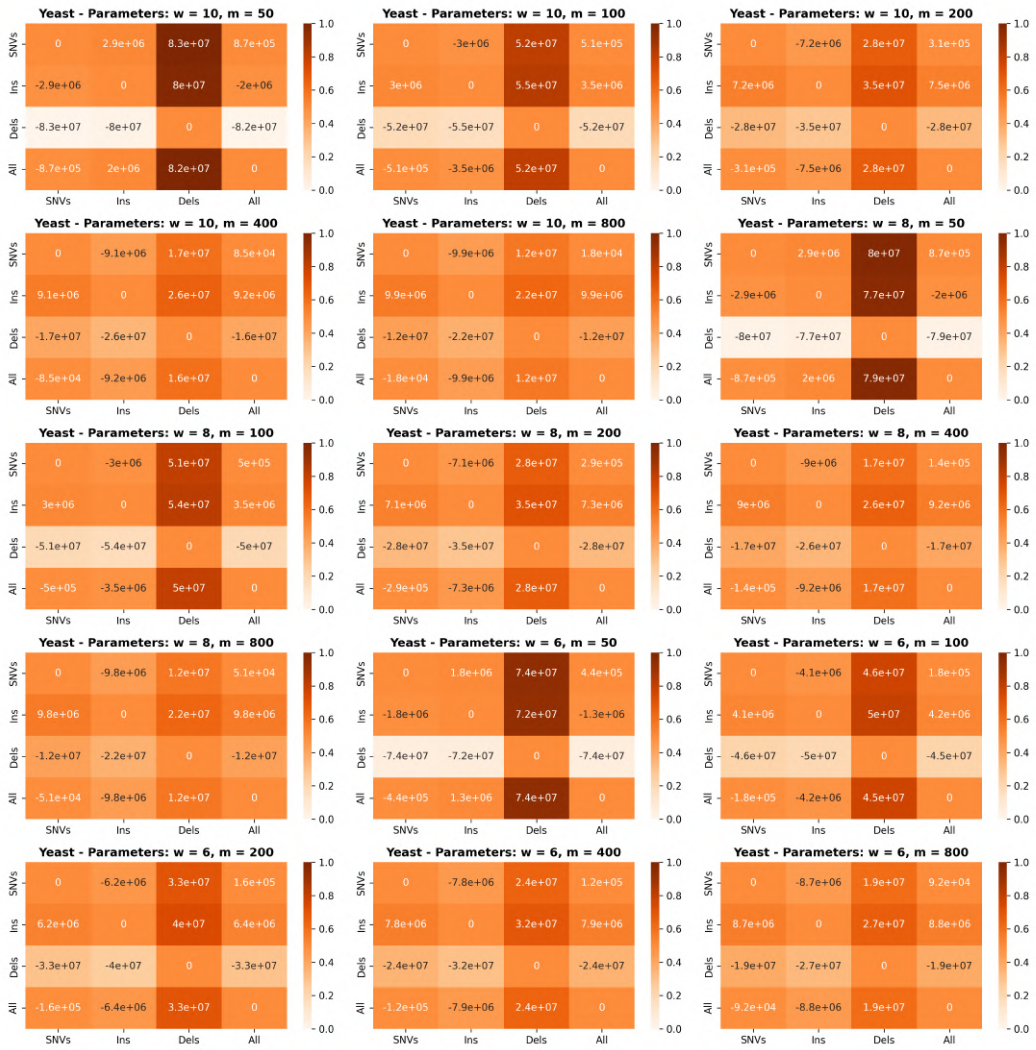


FIGURE B.7 Effect of probability of change 1% on dictionary size - Overall normalization

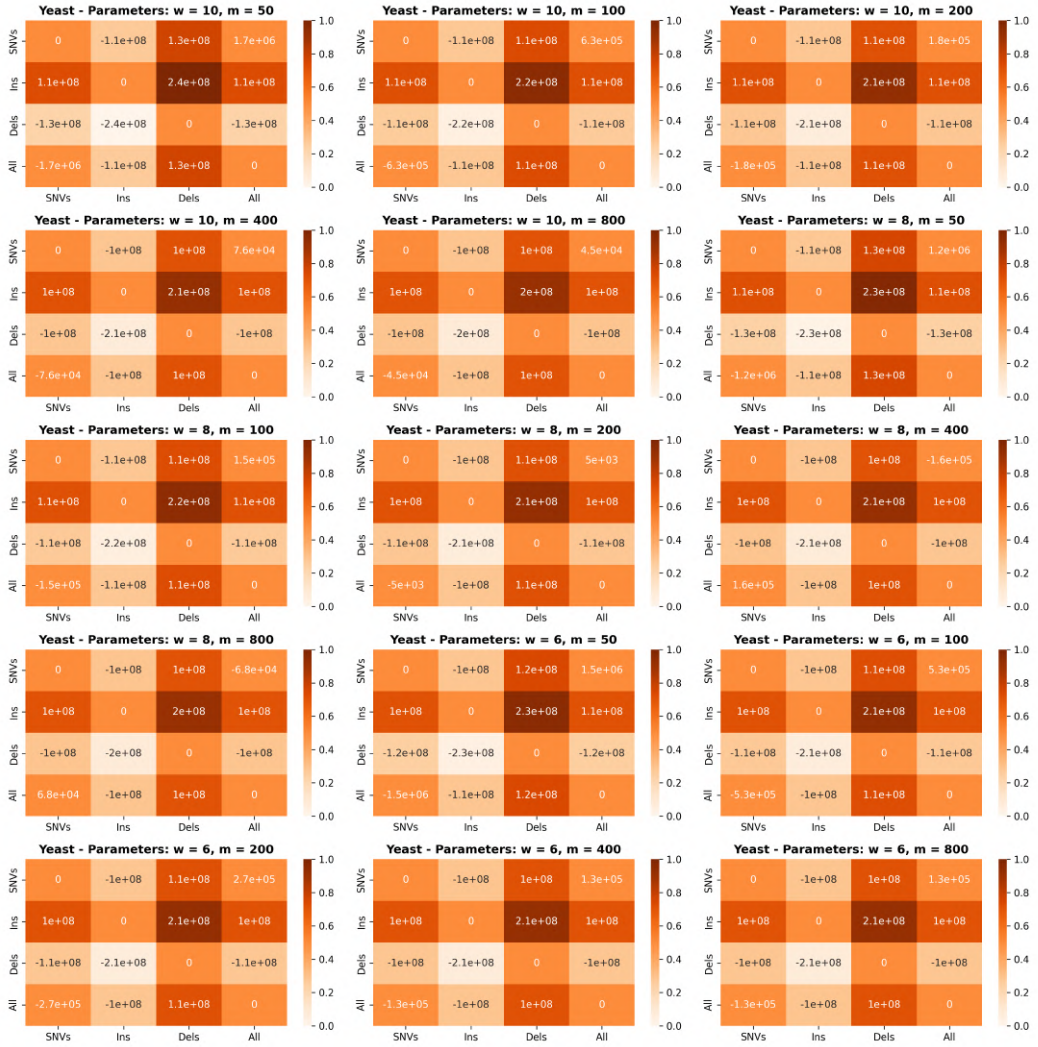


FIGURE B.8 Effect of probability of change 10% on dictionary size - Overall normalization

B.2 *S. enterica*

FIGURE B.9 Effect of probability of change 0.01% on dictionary size - Absolute value and single normalization

FIGURE B.10 Effect of probability of change 0.1% on dictionary size - Absolute value and single normalization

FIGURE B.11 Effect of probability of change 1% on dictionary size - Absolute value and single normalization

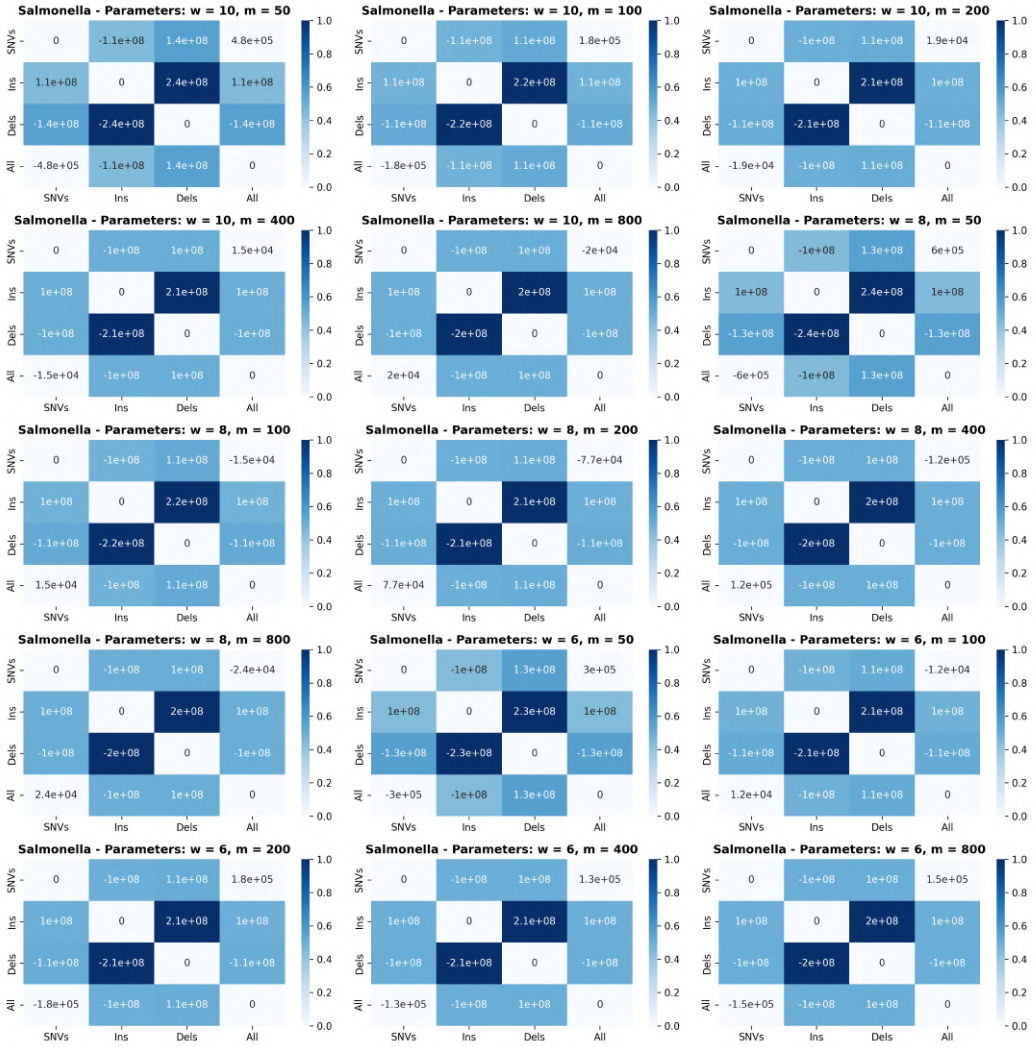


FIGURE B.12 Effect of probability of change 10% on dictionary size - Absolute value and single normalization

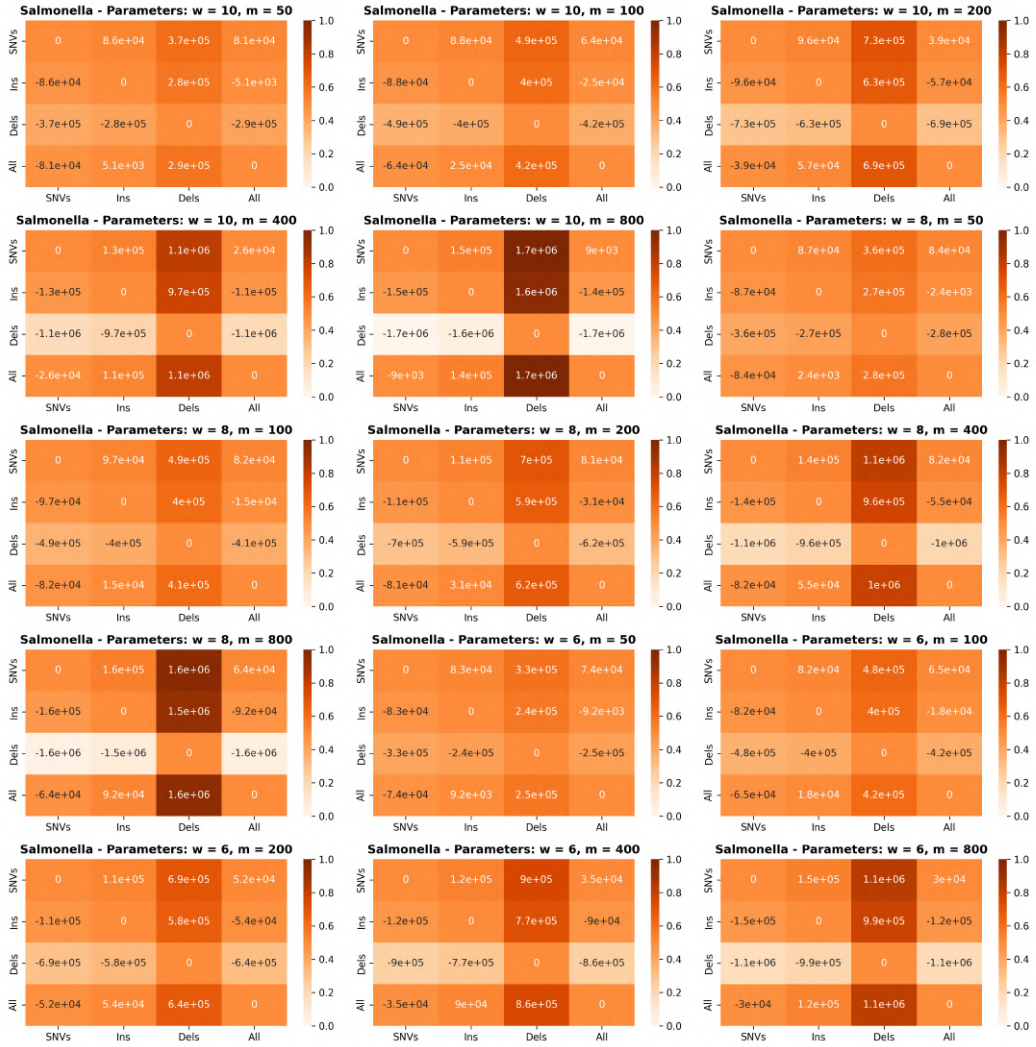


FIGURE B.13 Effect of probability of change 0.01% on dictionary size - Overall normalization

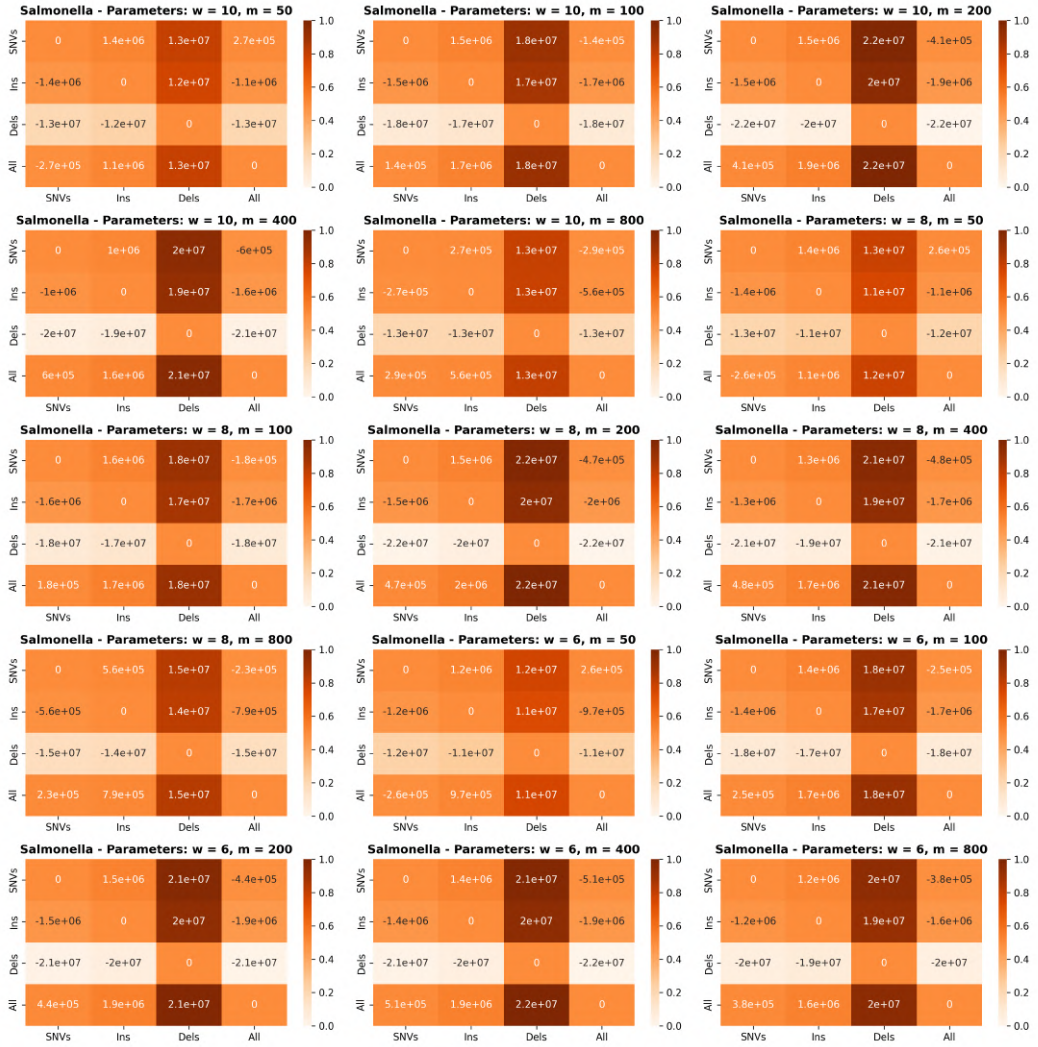


FIGURE B.14 Effect of probability of change 0.1% on dictionary size - Overall normalization

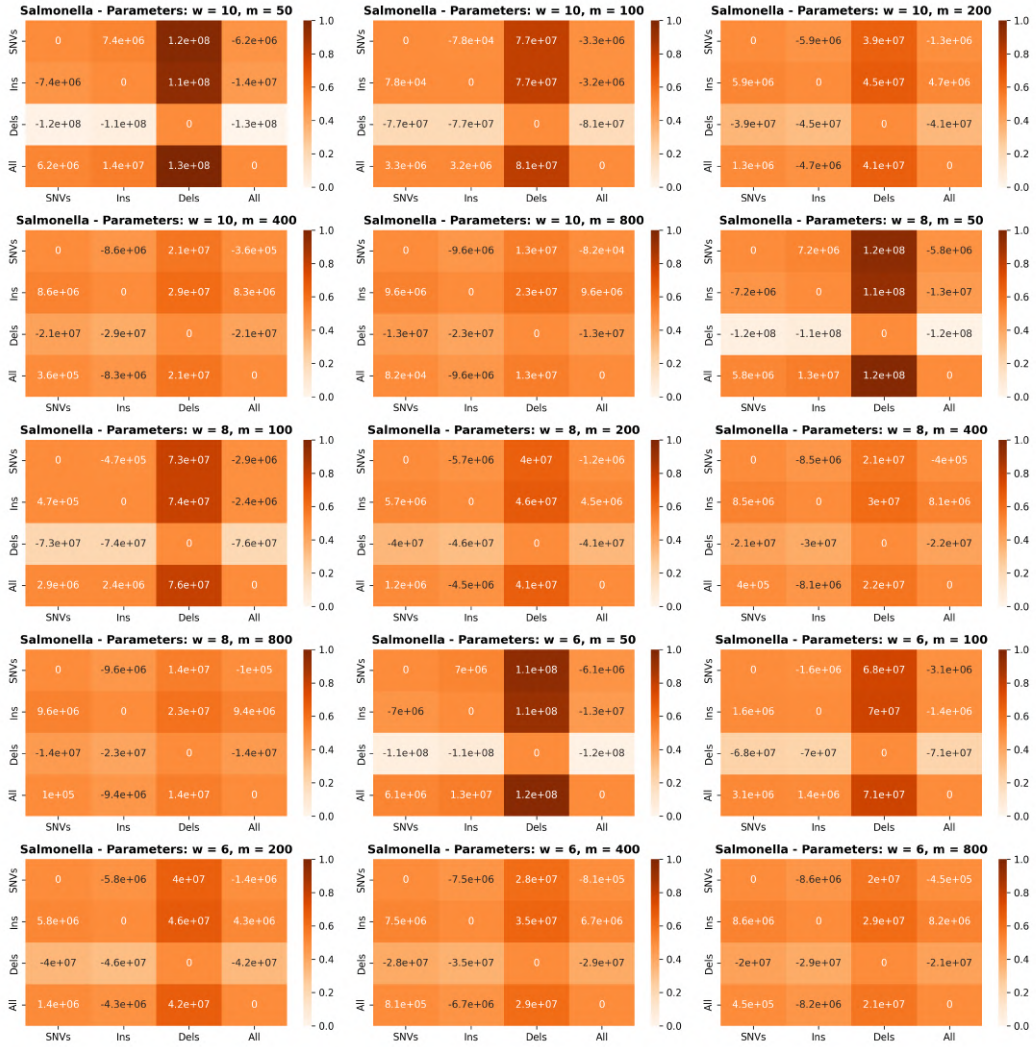


FIGURE B.15 Effect of probability of change 1% on dictionary size - Overall normalization

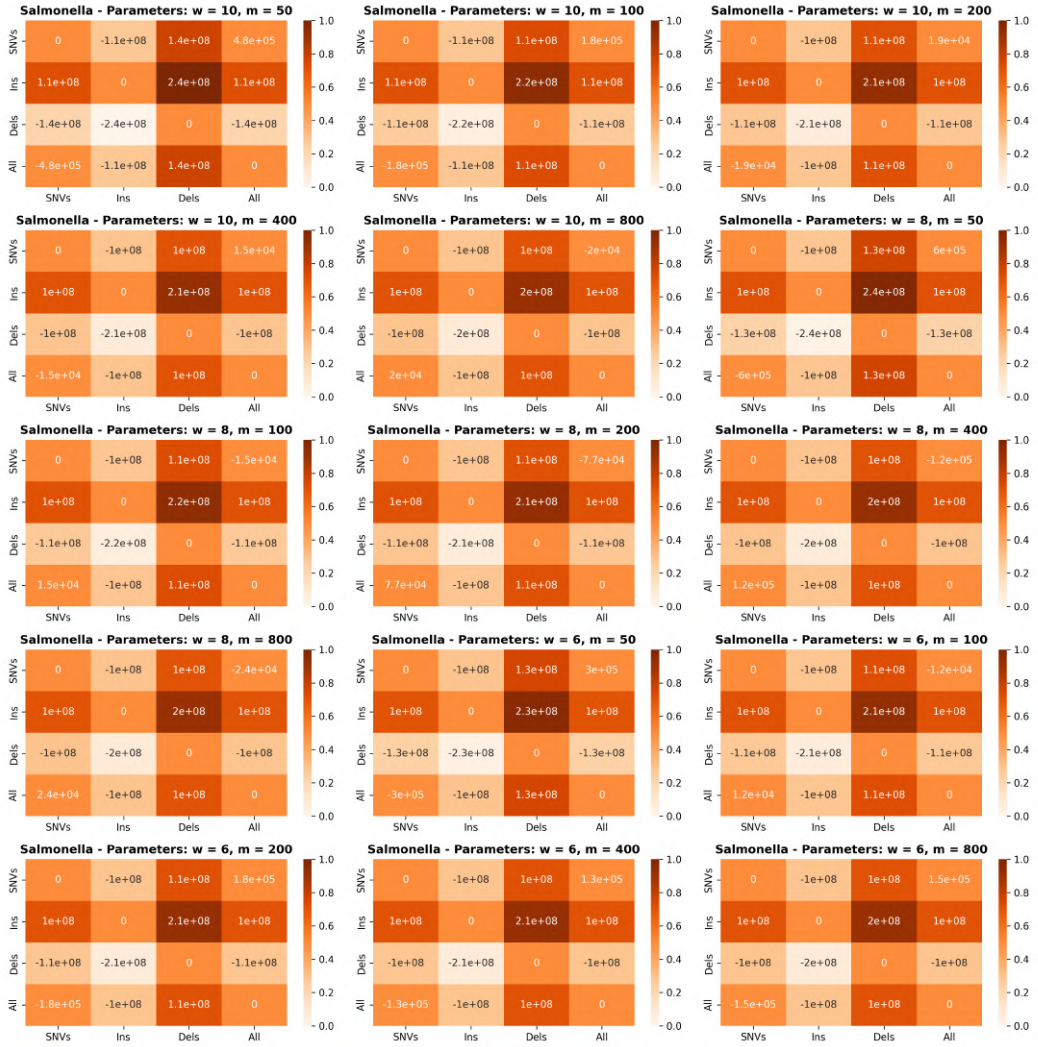


FIGURE B.16 Effect of probability of change 10% on dictionary size - Overall normalization

B.3 *SARS-CoV-2*

FIGURE B.17 Effect of probability of change 0.01% on dictionary size - Absolute value and single normalization

FIGURE B.18 Effect of probability of change 0.1% on dictionary size - Absolute value and single normalization

FIGURE B.19 Effect of probability of change 1% on dictionary size - Absolute value and single normalization

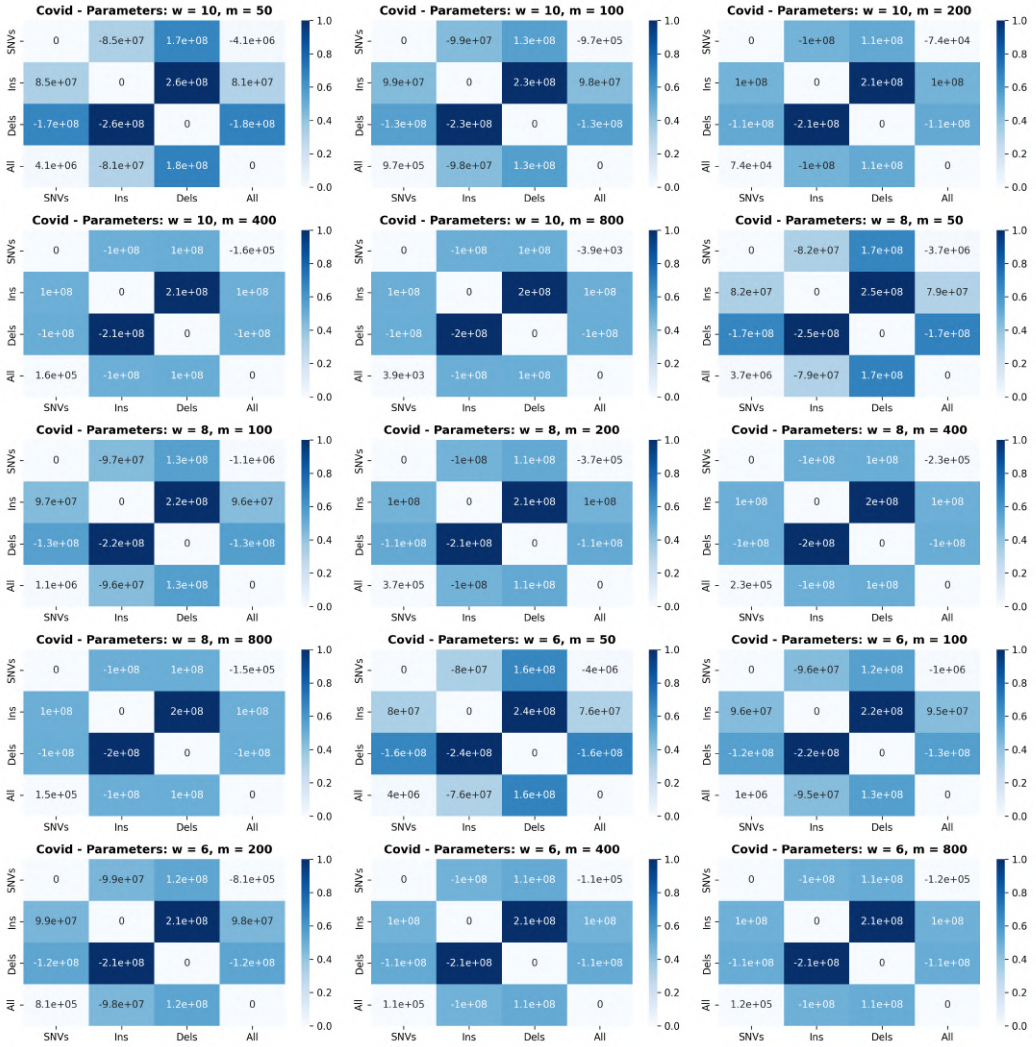


FIGURE B.20 Effect of probability of change 10% on dictionary size - Absolute value and single normalization

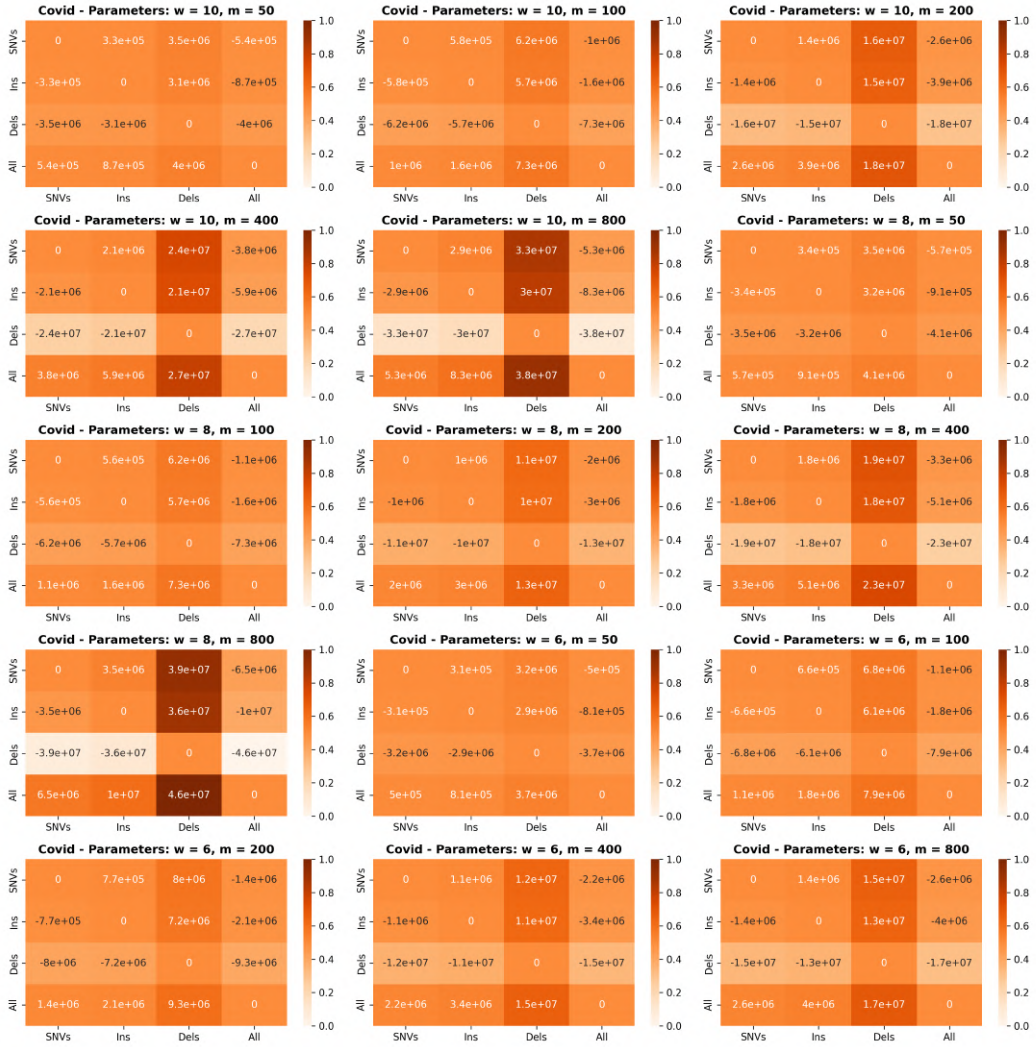


FIGURE B.21 Effect of probability of change 0.01% on dictionary size - Overall normalization

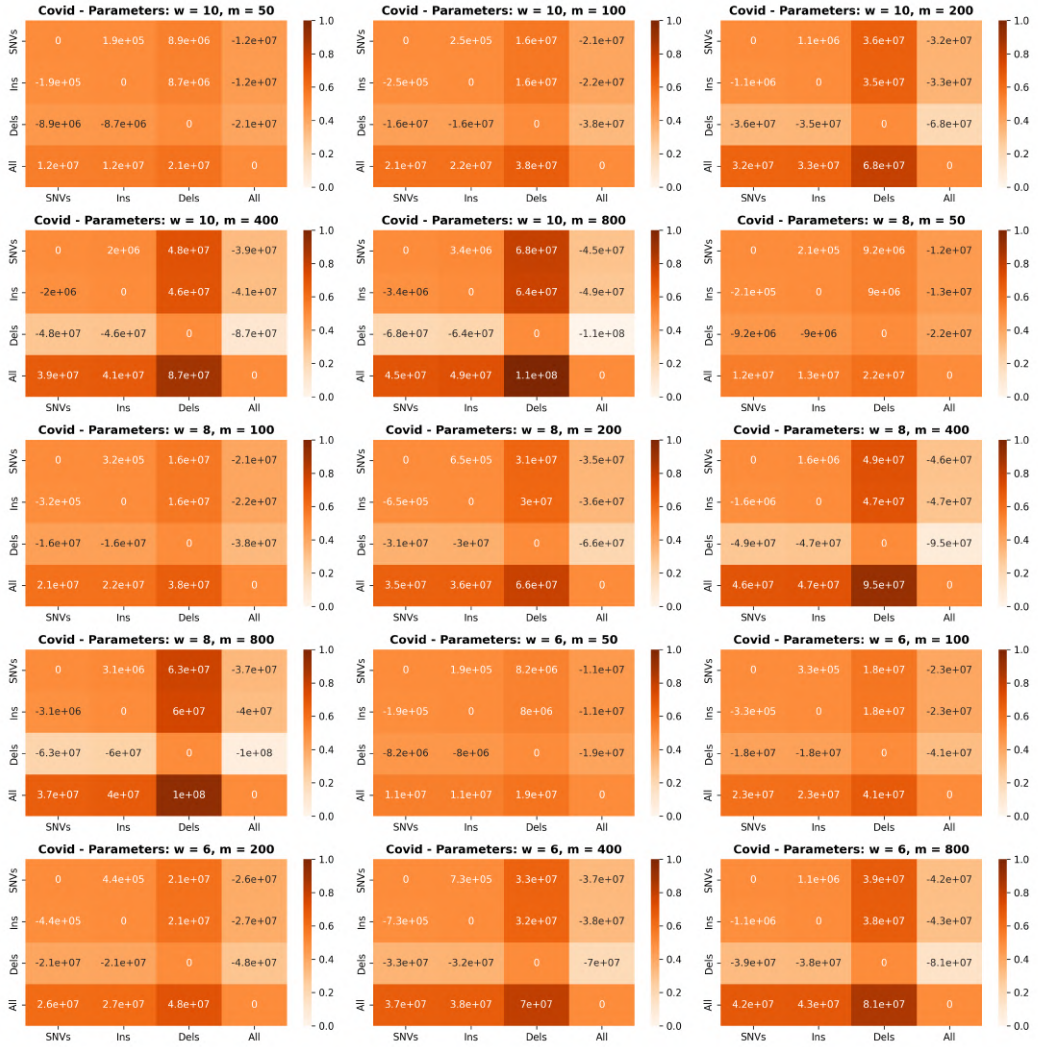


FIGURE B.22 Effect of probability of change 0.1% on dictionary size - Overall normalization

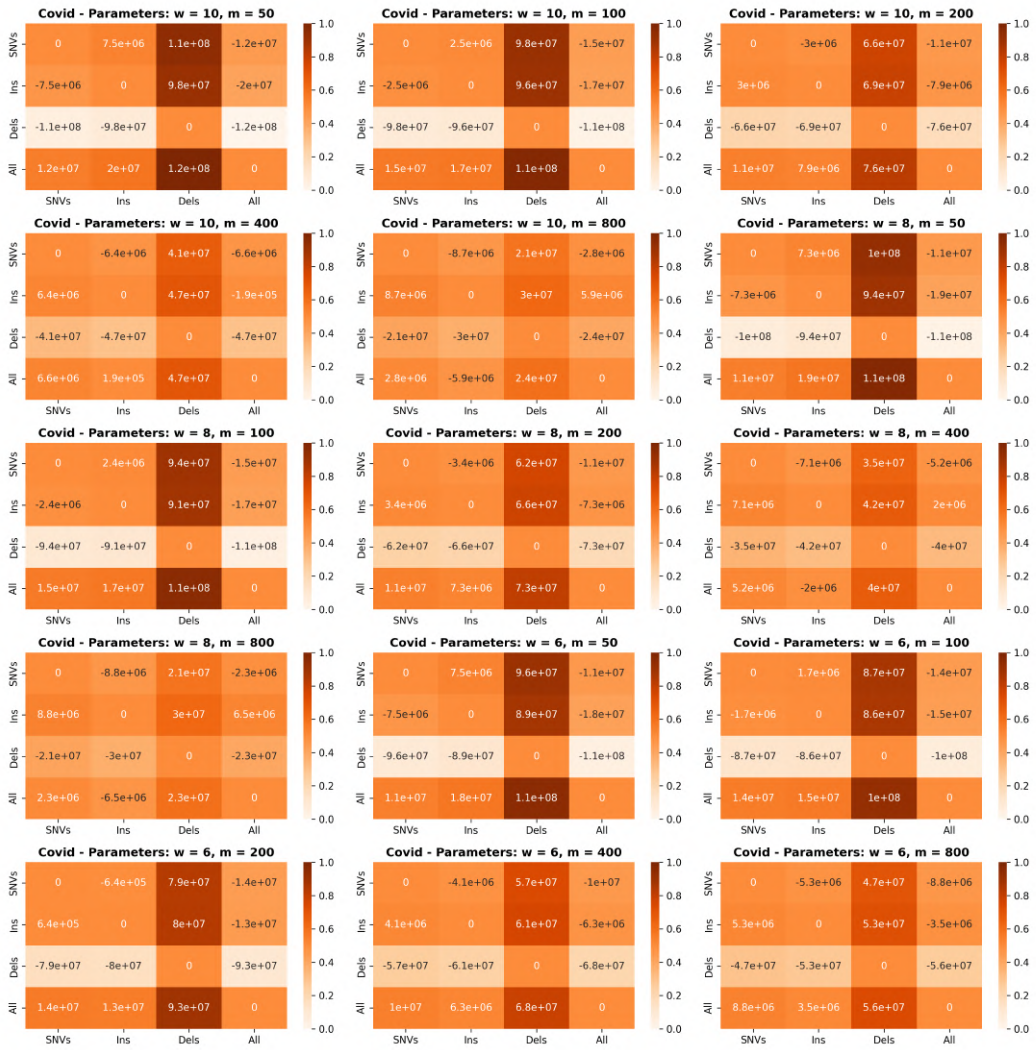


FIGURE B.23 Effect of probability of change 1% on dictionary size - Overall normalization

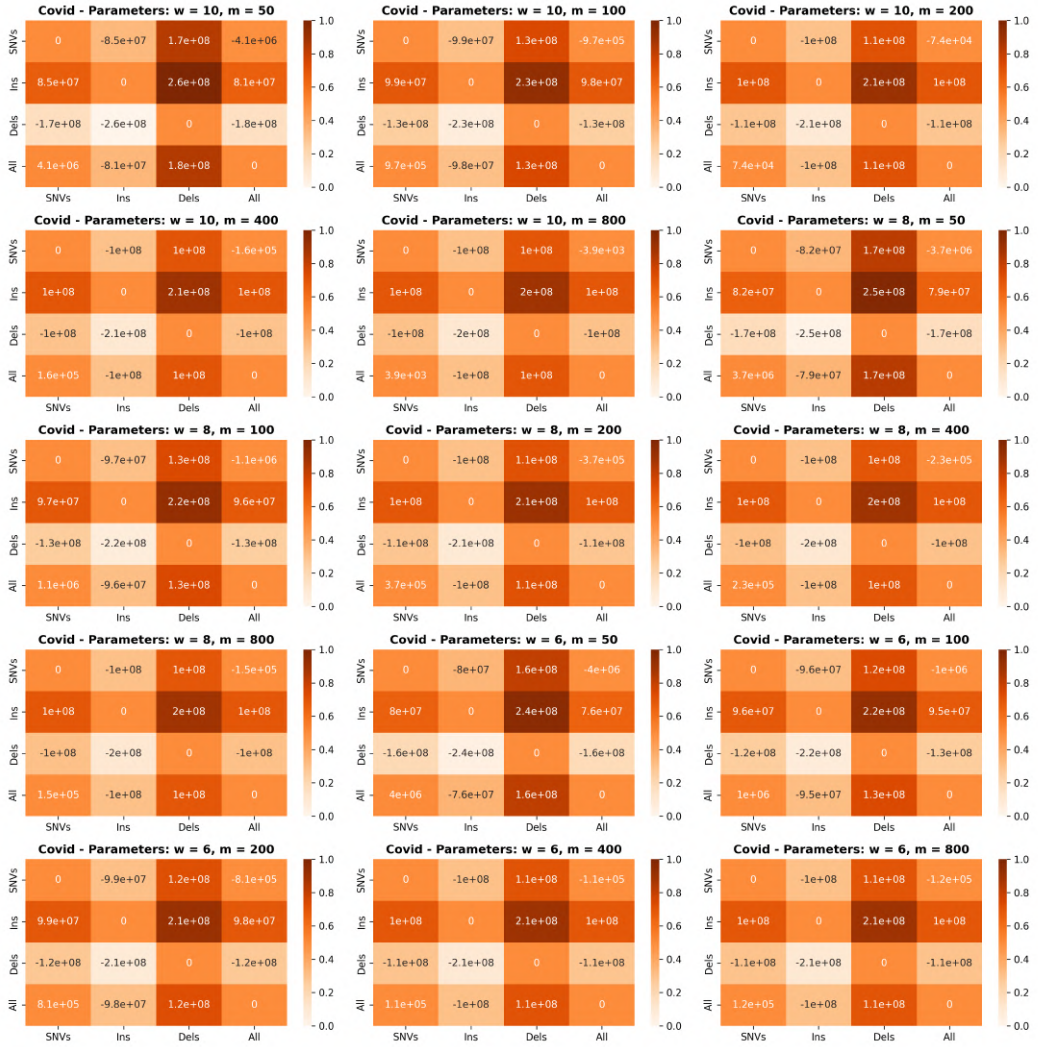


FIGURE B.24 Effect of probability of change 10% on dictionary size - Overall normalization

Bibliography

Allard, Marc W et al. (2016). “Practical value of food pathogen traceability through building a whole-genome sequencing network and database”. In: *Journal of clinical microbiology* 54.8, pp. 1975–1983.

Boucher, Christina et al. (2019). “Prefix-free parsing for building big BWTs”. In: *Algorithms for Molecular Biology* 14.1, pp. 1–15.

Branchu, Priscilla, Matt Bawn, and Robert A Kingsley (2018). “Genome variation and molecular epidemiology of Salmonella enterica serovar Typhimurium pathovariants”. In: *Infection and Immunity* 86.8, pp. 10–1128.

Burrows, Michael and David Wheeler (1994). “A block-sorting lossless data compression algorithm”. In: *Digital SRC Research Report, Tech. Rep*.

DNA Sequencing Costs: Data (Dec. 2023). [Online; accessed 13. Dec. 2023]. URL: https://www.genome.gov/about-genomics/fact-sheets/DNA-Sequencing-Costs-Data.

Ferragina, Paolo and Giovanni Manzini (2000). “Opportunistic data structures with applications”. In: *Proceedings 41st annual symposium on foundations of computer science*. IEEE, pp. 390–398.

Ganguly, Arnab et al. (2020). “Fm-index reveals the reverse suffix array”. In: *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*.

Genomic Data Science Fact Sheet (Apr. 2022). [Online; accessed 14. Dec. 2023]. URL: https://www.genome.gov/about-genomics/fact-sheets/Genomic-Data-Science.

giovmanz / Big-BWT · GitLab (Feb. 2024). [Online; accessed 6. Feb. 2024]. URL: https://gitlab.com/manzai/Big-BWT.

Gómez-Carballa, Alberto et al. (2020). “Mapping genome variation of SARS-CoV-2 worldwide highlights the impact of COVID-19 super-spreaders”. In: *Genome Research* 30.10, pp. 1434–1448.

Gonnet, Gaston H, Ricardo A Baeza-Yates, and Tim Snider (1992). “New Indices for Text: Pat Trees and Pat Arrays.” In: *Information Retrieval: Data Structures & Algorithms* 66, p. 82.

Hunter, J. D. (2007). “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3, pp. 90–95.

Karp, Richard M and Michael O Rabin (1987). “Efficient randomized pattern-matching algorithms”. In: *IBM journal of research and development* 31.2, pp. 249–260.

libsais (Feb. 2024). [Online; accessed 6. Feb. 2024]. URL: https://github.com/IlyaGrebnov/libsais.

Mäkinen, Veli et al. (2015). *Genome-scale algorithm design*. Cambridge University Press.

Manber, Udi and Gene Myers (1993). “Suffix arrays: a new method for on-line string searches”. In: *siam Journal on Computing* 22.5, pp. 935–948.

Muller, Ludo AH and John H McCusker (2011). “Nature and distribution of large sequence polymorphisms in S accharomyces cerevisiae”. In: *FEMS yeast research* 11.7, pp. 587–594.

Turnbull, Clare et al. (2018). “The 100 000 Genomes Project”. In: *BMJ: British Medical Journal* 361.

Waskom, Michael L. (2021). “seaborn: statistical data visualization”. In: *Journal of Open Source Software* 6.60, p. 3021.

Weiner, Peter (1973). “Linear pattern matching algorithms”. In: *14th Annual Symposium on Switching and Automata Theory (swat 1973)*. IEEE, pp. 1–11.

Acknowledgements

I wanted to dedicate this page to the people who contributed to the production of this thesis.

Firstly, I wanted to thank my supervisors, Zsuzsanna Lipták and Francesco Masillo, who supported and guided me every step of the way with great patience, kindness and professionality.

I sincerely thank my parents for giving me the opportunity to study in Verona and for supporting me even when the future was uncertain.

Thanks to all the people I met during this journey. In particular, I want to thank the group from CV2 and the one from Granada: it's also thanks to all of you that I'm here now writing all of this.

Thanks also to my flatmate Anna, who witnessed the best and the worst but was always there to support me.

Last but not least, I want to thank myself for pushing through all the doubts and difficulties. To Martina from the past: this is proof that not giving up was the right choice.