Fast mapping and querying over large scale typing data

Ana Correia
ana.duarte.correia@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2018

Abstract
High-Throughput DNA Sequencing (HTS) methods gave rise to a paradigm shift in microbial typing and genomic population structure studies. The ability to partially sequence the genomes of hundreds to thousands of strains created the need for effective ways to represent relationships between strains. Single Nucleotide Polymorphism (SNP) analysis and whole or core genome MultiLocus Sequence Typing (wgMLST or cgMLST), result in profiles that have thousands of loci which can be used for outbreak investigation, epidemiological surveillance of clones of interest and bacterial population or evolutionary studies. The first step to define these profiles is to map reads obtained through genome sequencing, identify relevant genes, and query existing typing databases to find if the strain being analyzed has been identified already, or if it is a new strain. Given the size of existing typing databases, the data volume resulting from HTS, and the urgency of these analyses, namely when in presence of outbreaks, the inherent computational problem of mapping and querying typing data has become a big challenge. To solve this issue, this work intend to demonstrate and proof a new approach that relies on Linear Codes, specifically on Reed Muller codes.

Keywords: Single Nucleotide Polymorphism, Hamming Distance; Linear Codes; Reed-Muller codes; approximate string matching.

1. Introduction
For many years, researchers have combined experimental biology with analytical methods, including algorithms. More recently, computational models have also been used to assist scientific communities in the interpretation of their experiments and results.

Molecular typing [3] is a way of identifying specific strains of microorganisms, such as bacteria or viruses, by looking at their genetic material. It is used to differentiate microorganisms at the subspecies, or strain level, for epidemiological investigations, infection control, public health and environmental sampling. DNA sequence-based typing methods require authoritative databases that link sequence variants to some nomenclature in order to facilitate communication and comparison between identified types in national or global settings.

2. Problem
High-throughput sequencing methods [18] gave rise to a paradigm shift in microbial typing and genomic population structure studies. The ability to partially sequence the genomes of hundreds to thousands of strains created the need for effective ways to represent relationships between strains. Single Nucleotide Polymorphism [9] (SNPs) analysis and whole or core genome MultiLocus Sequence Typing [10] (wgMLST or cgMLST), result in profiles that have thousands of loci which can be used for outbreak investigation, epidemiological surveillance of clones of interest and bacterial population or evolutionary studies. The first step to define these profiles is to map reads obtained through genome sequencing, identify relevant genes, and query existing typing databases to find if the strain being analyzed has been identified already, or if it is a new strain. Given the size of existing typing databases, the data volume resulting from high-throughput sequencing, and the urgency of these analyses, namely when in presence of outbreaks, the inherent computational problem of mapping and querying typing data has become a big challenge. When considering SNP data, the problem can be defined as follows:

**Problem 2.1.** Given a set of SNPs $S$, a SNP pattern $p$, over an alphabet $\Sigma = \{0, 1\}$, and a $k \in \mathbb{N}$, the goal is to find all approximate occurrences of $p$ in $S$ such that $\delta_{\text{Hamming}}(S_i, p) \leq k$, where $\delta_{\text{Hamming}}$ is the Hamming Distance.

One way to solve this problem is to use string approximation algorithms [14], that performs string
matching of a pattern in a text, while allowing errors to occur between the pattern and the text, with the goal of discovering the position of the pattern in the text. Approximate string matching can also discover what are the errors present between the pattern and the text. Approximate string matching is used in several different areas, like, signal processing, retrieval of information/text, pattern recognition and data mining.

The purpose of this paper is to demonstrate a new approach using linear codes, Reed Muller code, and to evaluate its computational complexity and performance.

3. Reed Muller
Reed Muller codes are some of the oldest error correcting codes. Error correcting [12] codes are very useful in sending information over long distances or through channels where errors might occur in the message. They have become more prevalent as telecommunications have expanded and developed a use for codes that can self-correct. Reed Muller codes were invented in 1954 by D. E. Muller and I. S. Reed [13] [17]. In 1972, a Reed Muller code was used by Mariner 9 to transmit black and white photographs of Mars. Reed Muller codes are relatively easy to decode, and first-order codes are especially efficient.

3.1. Definition of Terms and Operations.
The vector spaces used in this paper consist of strings of length \(2^m\), where \(m\) is a positive integer, of numbers in \(F_2 = \{0, 1\} \). The codewords of a Reed Muller code form a subspace of such a space. Vectors can be manipulated by three main operations: addition, multiplication, and the dot product. For two vectors \(x = (x_1, x_2, \ldots, x_n)\) and \(y = (y_1, y_2, \ldots, y_n)\), addition is defined by

\[
x + y = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n),
\]

The addition of a scalar \(a \in F_2\) to vector \(x\) is defined by \(a + x = (a + x_1, a + x_2, \ldots, a + x_n)\).

The complement \(\overline{x}\) of a vector \(x\) is the vector equal to \(1 + x\).

Multiplication is defined by the formula,

\[
x \cdot y = (x_1 \cdot y_1, x_2 \cdot y_2, \ldots, x_n \cdot y_n),
\]

The multiplication of a constant \(a \in F_2\) to vector \(x\) is defined by

\[
a \cdot x = (a \cdot x_1, a \cdot x_2, \ldots, a \cdot x_n).
\]

Let \(0 \leq r \leq m\). The \(r\)-th order Reed-Muller code \(R(r,m)\) is the set \(p\) of all binary strings of length \(n = 2^m\) associated with the Boolean polynomials \(p \equiv \{x_1, x_2, x_4, \ldots, x_m\}\) of degree at most \(r\). Consequently, the \(0\)-th order Reed-Muller code \(R(0,m)\) consists of the binary strings associated with the constant polynomials \(0\) and \(1\). This code is the repetition code of length \(2^m\), \(R(0, m) = \{0^m, 1^m\} = \{0000000, 1111111\}\). The other extreme situation is the \(m\)-th order Reed-Muller code \(R(m,m)\), consisting of all binary strings of length \(2^m\), that is, \(R(m,m) = F_2^n\), where \(n = 2^m\). The number of code-words can be found easily from the count of binary monomials in \(R(r,m)\) of degree at most \(r\). There are

\[
k = 1 + \binom{m}{1} + \binom{m}{2} + \binom{m}{3} + \ldots + \binom{m}{r}
\]

such monomials, and so there are \(2^k\) linear combinations of these. It is obvious that the closer \(r\) is to \(m\) the more code words there are. The \(r\)-th order Reed-Muller code \(R(r,m)\), has the following properties:

- Length of codewords: \(2^m\)
- Number of codewords: \(2^m\)
- Minimum distance between codewords: \(2^m - r\)
- Alphabet size: 2
- Rate: \(k/2^m\)

3.2. Encode
To define the encoding matrix of \(R(r,m)\), let the first row of the encoding matrix be \(11\ldots1\) (the vector with length \(2^m\) with all entries equal to 1). If \(r\) is equal to 0, then this row is unique in the encoding matrix. On the other hand, if \(r\) is equal to 1, then we add \(m\) rows corresponding to the vectors \(x_1, x_2, \ldots, x_m\) and to the \(R(0,m)\) encoding matrix. Thus, in order to form an \(R(r,m)\) encoding matrix, where \(r\) is greater than 1, we have to add \(\binom{m}{r}\) rows to the \(R(1,m)\) encoding matrix. These added rows consist of all the possible reduced degree \(r\) monomials that can be formed using the rows \(x_1, x_2, \ldots, x_m\).

Example 3.1. \(R(1,3)\)
When \(m = 3\) we have:

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}
\]

The number of rows of these encoding matrices is \(k = 1 + \binom{m}{1} + \binom{m}{2} + \binom{m}{3} + \ldots + \binom{m}{r}\). So, the sent message must be in blocks of length \(k\). Let \(m = (m_1, m_2, m_3, \ldots, m_k)\) be such a block. Then the encoded message \(\mathcal{M}\) is the \(\sum_{i=1}^{k} m_i R_i\), where \(R_i\) indicates the rows of the encoding matrix of \(R(r,m)\).
Example 3.2. Encoding with $R(1,3)$

Using $R(1,3)$ to encode $m = (0111)$ gives:

$$0*(11111111)+$$
$$0*(11110000)+$$
$$1*(11001100)+$$
$$1*(10101010)+(01100110)$$

3.3. Decode

The decoding method checks which row $R_i$ of the encoding matrix was used to form the encoded message. The implementation of this method requires the use of characteristic vectors of the encoding matrix rows. In order to find the characteristic vector, we work on the monomial $r$ associated with the row of the matrix. After that, we take the set of all $x_i$ that are not in $r$, but only in the encoding matrix. The characteristic vectors are those vectors that correspond to monomials $x_i \tau_r$, such that exactly one of $x_i$ or $\tau_r$ belongs to each monomial for all elements of the set of all $x_i$. The dot product of these characteristic vectors with all the rows of the used code matrix yields 0, except the row to which the vector corresponds.

This method is precisely described in the following three steps of an algorithm [1]:

1. Choose a row of the given encoding matrix code and find $2^{n-r}$ characteristic vectors for that row. Then, form the dot product of these vectors with the encoded message.

2. Compute the majority value (either 1 or 0) of the dot products, and assign it to each row.

3. Executing steps 1, 2 from the bottom of the matrix to the top, multiply the majority value assigned to each row by its corresponding row. Add the results altogether, and then sum this up to the received encoded message. If there is a majority of 1’s in the resulting vector, then assign 1 to the top row. Otherwise, if there is a majority of 0’s, then assign 0 to the top row. Adding the top row, multiplied by the assigned value, leads to the original encoded message. Using this algorithm, it is obvious that we can identify the errors occurred during the transmission of encoded message. The vector that is formed using the assigned values of each row, from the top row all the way to the bottom row of the encoding matrix, is the original message.

Example 3.3. Decoding Using $R(1,3)$

Assuming an original message $m = (0110)$, using the $R(1,3)$ encoded matrix we get the encoded message $M = (00111100)$. As it was already mentioned, the distance in this case is $2^{1-1} = 4$, and therefore, it can correct one error.

Assuming that, during message transmission, one error occurred at the first leftmost bit, the encoded message after the error is $M’ = (10111100)$. The characteristic vectors of the last row of the encoded matrix are $x_1 x_2, x_1 x_2, x_1 x_2$ and $x_1 x_2$. The vector related to $x_1$ is $(11110000)$, thus $\tau_1$ is $(00011111)$. Similarly, $x_2$ is $(11001100)$, and thus $\tau_2$ is $(00110011)$. Therefore, $x_1 x_2$ is $(11000000)$, $x_1 x_2$ is $(0011000000)$, $x_1 x_2$ is $(00001100)$ and $x_1 x_2$ is $(00000011)$. Computing the dot product of these vectors with $M’$, we get the values 1, 0, 0, 0 respectively, leading to majority value of 0 for $x_1$. Repeating the process for the second to last row of the matrix, we get the values 0, 1, 1, 1 respectively, leading to majority value 1 for $x_2$. Working similarly, we conclude that the coefficient of $x_1$ is also 1.

Adding $(0(10101010)+1(11001100)+1(11100100)+1(11011010)\text{ and } 0(11101010)\text{ and } 1(11110000)\text{ we get } M'' = (00111100)$. Then, we notice, that adding $M’$ and $M''$ we get $(10000000)$, which has more 0’s than 1’s, leading to 0 for the coefficient of the first row of the used matrix. Putting together the four coefficients that correspond to four rows 0, 1, 1, 0 we get the original message. Additionally, we can determine the position of the error at the first leftmost bit.

4. Approach

The idea of this approach is to use the linear code with detection and correction of errors to create groups of SNPs that are very similar. A linear code is a vector subset with $2^n$ elements of the vector space made up of all $2^n$ with $n$ elements. All elements of this subset have a minimum Hamming distance $d$ between them. This type of code can simultaneously detect up to $d-1$ errors and correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors.

In this approach, we consider that all SNPs in the dataset are possible code words of the linear code. For each SNP it checks if it is a code word with errors and obtains the true codeword. With this indexing to the code space of the linear code you can organize the SNPs of the dataset in the regions of each codeword. To find the neighbors of an SNP query with $k$ errors, it is necessary to find the code words that contain these neighbors.

We generate the neighborhood of the SNP query and for each neighbor we verify which code word is associated with it and filter itself all the SNPs that belong to its region that are close to the SNP query.

As can be seen in Figure 1, a code word may contain several SNPs that belong to the neighborhood of the SNP query so these neighbors will not be processed because they have already been found.
This approach consists of two sub-approach’s: indexing and query.

4.1. Indexing Sub-Approach
The indexing sub-approach is responsible for mapping all SNPs in the dataset to the Reed Muller code space. To perform this mapping, you must perform the following operations:

- **Decode**: this operation will detect and correct the possible errors that the SNP may contain and return the possible message that in the concept of Communication Theory may have been sent to the channel.

- **Encode**: with the message the encode is performed and we get the correct code word.

This algorithm receives a set of SNPs, all with the same size. For each SNP, the decode and encode operations are performed and the corresponding code word is obtained. The next step is to verify that this code word already exists in the indexing structure, which in this case consists of a Map. If this code word already exists then the SNP is saved in the list of SNPs of the code word, otherwise, the new key is added to the Map (code word) and the new list already with the SNP added.

4.2. Search Sub-Approach
The search sub-approach has a responsibility to find all SNPs that are similar up to $k$ errors of an SNP query. This sub-approach has the following steps:

1. Generate the neighborhood up to $k$ SNP Query errors;

2. For each neighbor generated, the Reed Muller is used to obtain the code word of this (Decode and Encode);

3. If the code word exists in the indexing structure, the SNPs that are within the region of the code word that are similar to the SNP query are searched.

5. Results
To solve the problem presented in definition 2.1, it is necessary to go through all the SNPs of the dataset and calculate for each SNP the distance of Hamming with the SNP query and if it is smaller than $k$ then it saves itself this SNP in a list (naive approach).

The main objective of the experimental evaluation is to understand to which extent the new approach is competitive with the naive version. The new approach was implemented in C++ and compiled with g++ (GCC version 4.2) with full optimization enabled. We evaluated the proposed approach using both real and synthetic data sets. All synthetic sequences are binary sequences uniformly sampled. Results presented in this section were average over five runs.

All experiments were performed on a machine with the following specifications: Intel Core i7 2.6 GHz quad core processor with 16 GB 1600 MHz DDR3 memory.

In this section we will use the following notation to demonstrate the results of the experiments:

- $s$ for dataset size.
- $n$ for SNP length.
- $d$ for RM codewords distance.
- $k \in \mathbb{R}$ is number of errors.
- $r$ RM order.

5.1. Index sub-approach experiments
In these experiments we intend to evaluate the running time and RM distribution behavior, that is, the number of SNPs lying in a region of a code word but for which the Hamming distance to the code word is greater than the minimum distance of RM.

As can be seen in Figures 2 and 3, the execution time of this sub-approach is linear. The running time includes the time to read the dataset and, for each SNP in the dataset, perform the operations decode and encode, and finally add them to the data structure.

The indexing sub-approach uses $O(cs)$ space, where $s$ is the size of the dataset and $c$ is the number of words in the RM code.

Table 1 shows the distribution of the SNPs in the regions of the first and second order RM code words. As can be seen, for second order RM, there are more code words with SNPs where the distance of each SNP to its code word is higher with the minimum distance for the RM, because the minimum distance from the $RM(2,m)$ is less than $RM(1,m)$. Many of the RM code words $RM(2,m)$ contain only one element. From the results of this experiment, it was decided to use the $RM(1,m)$ in our approach.

![Figure 1: Main Idea. The variable $d$ is the minimum distance between codewords of RM.](image)
5.2. Search sub-approach experiments

In these experiments we intend to evaluate the execution time and the precision. For precision, we will count:

- **True positives** - Number of SNPs that were returned by the approach with a similarity equal or lower than $k$.
- **False positives** - Number of SNPs that were returned by the approach with similarity above $k$.

<table>
<thead>
<tr>
<th>$m$</th>
<th>SNP Size</th>
<th>Total codewords</th>
<th>$\delta_H(c,s) &gt; d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>16</td>
<td>32</td>
<td>25%</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>64</td>
<td>25%</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>128</td>
<td>24%</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>256</td>
<td>22%</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>512</td>
<td>20%</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>837</td>
<td>44%</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>1076</td>
<td>54%</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>1087</td>
<td>91%</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>1128</td>
<td>99%</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>1211</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 1: Distribution of SNPs for $1^{st}$ order and $2^{nd}$ order RM for different synthetic datasets with different sizes of SNPs ($m$). All synthetic datasets contain 2000 elements.

It will also be demonstrated the comparison of this approach with the naive version.

Table 2 shows the running time and the number of comparisons for the initial approach and the naive version, with $n = 16$ and where the $RM(1,4)$ was used with a distance of $d = 8$, and also with $n = 32$ and where $RM(1,5)$ was used with distance $d = 16$. As can be seen, the approach compares fewer SNPs than the naive version to $k \leq \frac{d}{2}$, which is the error-correction capability of RM. But can only get better running time than the naive version for $k = 1$.

6. Discussion

This approach has the advantage of not needing to create/rebuild a new data structure whenever we want to add a new SNP (that does not belong already to the dataset). To add a new SNP, we need only to perform the decode and encode operations and add it to the data structure. The algorithms that could be used to solve the problem [2] [14, 15] [23] [16] rely often on static data structures and, if we want to add a new element to the data structure, then we will need to rebuild the data structure from scratch. In the previous section we observed that our approach compares fewer SNPs than the naive version, especially for $k = \frac{d}{2}$, which was expected because the RM can correct errors up to $\frac{d}{2}$. But does not get better execution time than the naive version. The reason for this behaviour is the processing time required for the neighborhood generation and the processing time for the decode and encode operations. This is because the neighborhood of the SNP query for the different values of $k$ contains many SNPs that can be valid neighbors.

A possible solution is between we get a code word, we must generate a neighbor that does not belong to the same code region. In this way, neighbors are generated that belong to other regions of the code that have not yet been processed.

This solution is a pseudo-boolean problem [11], also known as 0-1 integer linear programming (0-1 ILP), consists of determining if a set of PB-constraints [19] are satisfiable or not. Note that this is an NP-complete problem [4]. If there exists a model which evaluates all PB-constraints to true, the set is considered satisfiable.

7. Conclusion

In this paper a new approach was presented to solve problem 2.1. This new approach uses linear code with correction and detection errors, Reed Muller, to group the SNPs that belong to a region of a code word. The search for the neighboring SNPs of an SNP query is based on finding all the code words that contain in their region neighbors valid.

In this paper we also present the results of the
experiments and compared the results obtained with the results of the naive version. As noted, the new approach does not get better execution time than the naive version, but it can compare fewer SNPs during the neighbors search than the naive version, especially for a $k = \frac{d}{2}$ of the code.

The main contribution of this paper was to demonstrate a different way to solve problem 2.1 without using the classical methods of approximate string matching.

8. Future Work

As demonstrated in section 5, the approach presented in this paper has two main problems that make it not able to have better results than the naive version presented in Section 4, which are:

1. Running time of Redd-Muller decode and encode operations.

2. Generation of the neighborhood of the SNP query.

To solve the first problem, it is necessary to find and develop an RM decode algorithm that is faster to process than the algorithm that was used and presented in Section 3.

Since the creation of RM codes, researchers and students have attempted to improve their encode and decode and extend their error correction and detection properties [5, 7, 20, 21, 8, 22, 6]. In the article [22] authors present a list of possible decoding algorithms for RM. A future work would be to test these algorithms and evaluate what would be the best one to use in the proposed approach.

The extension of the approach presented in Section 6 minimizes the number of neighbors generated that are in the neighborhood of the SNP query, but still generates neighbors that are not in the index structure, that is, neighbors for which their code words do not exist in the indexing structure.

A possible solution to this problem would be to generate all the code words that do not contain elements in their region and to introduce for each code word a PB-constraint to indicate to the neighborhood generator to not generate neighbors that are in the regions of those code words.

References


---

<table>
<thead>
<tr>
<th>SNP Size</th>
<th></th>
<th>Naive</th>
<th></th>
<th>Approach</th>
<th></th>
<th>True Negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bits</td>
<td>1</td>
<td>1.57</td>
<td>2000</td>
<td>0.45</td>
<td>158</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1.56</td>
<td>2000</td>
<td>3.44</td>
<td>254</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.57</td>
<td>2000</td>
<td>61.77</td>
<td>258</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.63</td>
<td>2000</td>
<td>1186.63</td>
<td>1407</td>
<td>1193</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.54</td>
<td>2000</td>
<td>1801.98</td>
<td>2000</td>
<td>1752</td>
</tr>
<tr>
<td>32 bits</td>
<td>1</td>
<td>2.24</td>
<td>2000</td>
<td>2.12</td>
<td>157</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2.18</td>
<td>2000</td>
<td>46.71</td>
<td>241</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>2.16</td>
<td>2000</td>
<td>98993.68</td>
<td>1182</td>
<td>374</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2.29</td>
<td>2000</td>
<td>1212984.63</td>
<td>1695</td>
<td>498</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.45</td>
<td>2000</td>
<td>3320929</td>
<td>1895</td>
<td>622</td>
</tr>
</tbody>
</table>

Table 2: Running time and number of comparisons between dataset SNPs and SNP query (True + False Positives) for datasets with $s = 2000$, for $n = 16$ and $n = 32$. In these experiments, $RM(1, 4)$ and $RM(1.5)$ with distances of code $d = 8$ and $d = 16$. 


