Czech Technical University in Prague
Faculty of Information Technology

ASSIGNMENT OF MASTER'S THESIS

Title:	Searching for CRISPR segments using self-index
Student:	Bc. Ondřej Cvacho
Supervisor:	prof. Ing. Jan Holub, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Create a survey of tools for storing genomes and searching for CRISPR (Clustered regularly-interspaced short palindromic repeats) segments. Create a survey of current self-index data structures. Design and implement a self-index for collection of genomes allowing efficient search for CRISPR segments. Perform experimental evaluation of the implementation.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 17, 2016

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Master's thesis

Searching for CRISPR segments using self-index

Bc. Ondřej Cvacho

Supervisor: prof. Ing. Jan Holub, Ph.D.

9th May 2016

Acknowledgements

I would like to thank my thesis advisor prof. Jan Holub for his valuable and constructive suggestions during the planning and development of this research work. I would also like to thank my family and friends for their support and encouragement throughout my study. Special thanks should be given to my grandfather, Ing. Julius Cvacho, for being my motivation despite all his stories.

Declaration

In Prague on 9th May 2016 . . . .

Citation of this thesis

Cvacho, Ondřej. *Searching for CRISPR segments using self-index.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Práce se zaměřuje na využití kompaktních datových struktur v hledání CRISPR segmentů za použití self-indexů. Hledání CRISPR segmentů je srovnatelné s přibližným vyhledáváním řetězce za pomoci generování a vyhledávání všech podobných segmentů s možnou chybou až m. Navržené řešení se snaží redukovat takovou množinu všech podobných řetězců za použití De Bruijnových grafů. Experimentální vyhodnocení prokázalo možnost redukce až o čtyřnásobek.

Klíčová slova komprese, vyhledávání, bioinformatika, DNA, De Bruijn grafy, index, FM-index, kompaktní datové struktury

Abstract

The work focuses on the succinct data structures and their use for optimizing search for CRISPR segments using self-index. The search for segments uses approximate string matching by generating all possible segments with m or less mismatches and searching for each of them. I have proposed the solution to reduce the number of possible segments using the De Bruijn graphs as a filter. The experiments have shown that we can reduce the number of the segments almost four times.

Keywords compression, search, bioinformatics, DNA, De Bruijn graphs, index, FM-index, succinct data structures

Contents

Introduction **1**

1 Definitions **3**
- 1.1 Basic Definitions 3
- 1.2 Self-Index 4
- 1.3 Suffix Array 4
- 1.4 Burrows–Wheeler transform 6
- 1.5 FM-Index 8
- 1.6 Compressed Suffix Array 10
- 1.7 Succinct Data Structures 12
- 1.8 De Bruijn Graph 19

2 Bioinformatics **25**
- 2.1 DNA 25
- 2.2 RNA 25
- 2.3 FASTA 26
- 2.4 CRISPR 27

3 Existing Solutions **29**
- 3.1 Existing Libraries 29
- 3.2 Existing Applications 30
- 3.3 Compare of Existing Tools 35

4 Analysis and design **39**
- 4.1 Approximate String Matching 39
- 4.2 De Bruijn Graph 40
- 4.3 Design Proposal 42

5 Implementation **45**
- 5.1 Overall Structure 45

5.2 Settings and Input 46
5.3 Index 47
5.4 Off-target Generation 47
5.5 DBG Construction 48

6 Experiments **51**
6.1 About Experiments 51
6.2 Suffix Array 53
6.3 FM Index and BWT 54
6.4 De Bruijn graph 56
6.5 Generation of the Off-targets 56

Conclusion **63**

Bibliography **65**

A Data **71**
A.1 De Bruijn Graph 71
A.2 FM Index 73
A.3 Measurements 73

B Usage **77**
B.1 Installation 77
B.2 Usage 78

C Acronyms **79**

D Contents of enclosed CD **81**

List of Figures

1.1 Connection between the SA and the BWT for the text T. 7
1.2 Equations for updating borders of search interval 9
1.3 Wavelet tree of a string 18
1.4 De Bruijn graph 20
1.5 *De Bruijn graph* representation. 22
1.6 Equations for updating borders of search interval, *De Bruijn graph* 23

4.1 The number of strings with Hamming distance up to m from a string w 40
4.2 The size of the $H_5(w)$, $|w| = l$ 41
4.3 Number of nodes of the DBG for various k and input file size. . . . 42
4.4 Number of edges of the DBG for various k and input file size. . . . 43
4.5 Ratio of the number of edges to the number of nodes. 44

5.1 Structure of the application. 46

6.1 Time of suffix array construction using various algorithms. 53
6.2 Size of the input text and the resulted SA. 54
6.3 Time of the count query for RRR with the block size b. 55
6.4 Time of the construction of the succinct DBG for various k and input file size. 56
6.5 Size of the succinct DBG for various k and input file sizes. 57
6.6 Time to generate and score off-targets using *simple* method. 58
6.7 Number of generated off-targets by the *simple* method. 58
6.8 Number of generated off-targets by the DBG method and $k = 5$. . 59
6.9 Number of generated off-targets by the DBG and *simple* methods and $m = 5$. 59
6.10 Time to generate and score off-targets using *simple* and the DBG method, $m = 5$. 60

A.1 Number of generated off-targets by the DBG method and $k = 3$. . 73

A.2 Number of generated off-targets by the *DBG* method and $k = 4$. . 74
A.3 Number of generated off-targets by the *DBG* method and $k = 6$. . 74
A.4 Number of generated off-targets by the *DBG* method and $k = 7$. . 75

List of Tables

1.1 Summary of time & space complexity of SA operation 6
1.2 Burrows-Wheeler tranformation using Burrows-Wheeler matrix for string *banana* 7
1.3 FM-index example: data structures 9
1.4 Compressed Suffix Array using Ψ function 12
1.5 Compressed Suffix Array using sampling 12
1.6 The space and time results for *esp*, *recrank*, *vcode*, *sarray* and *darray* 17

2.1 Defined symbols of FASTA format. 26

3.1 Time comparison of the CRISPR/Cas9 design tools. 37

4.1 The size of the $N_m^H(w)$, $|w| = 20$ 40

6.1 Summary of informations about testing environment. 51
6.2 Input test files and corresponding chromosomes they were created of. 52
6.3 Time of locate query and the size of the index. 55
6.4 Total time of the *DBG* generator, $k = 5$ 60

A.1 The number of nodes of *DBG*. 71
A.2 The number of edges of *DBG*. 71
A.3 Ratio of the number of edges to the number of nodes. 72
A.4 The size of succinct *DBG* [MB]. 72
A.5 Construction time of *DBG* [s]. 72
A.6 Time of count query for *RRR* with block size b. 73

Introduction

Searching is one of the essential operations in the informatics. So, naturally we want the operation to be as fast as possible. The indexing techniques allow us to search in the text in almost constant time. However, the faster the index operations are the more space it takes. The compromises are made to balance between time and space requirements of the index of the text. The size of the index may be reduced by using the succinct representations. The succinct data structures allows us to reduce the size, but maintain the time of access operations of the original data structures.

Everything gets more complicated when we need to search with possible mismatches. This is the case, when we do not know exactly what to look for and only have the approximate information about what we want to find. However, the approximate string matching can be easily replaced by the exact matching by using the clever approach.

Recent technological advancements have provided a large amount of data. Among other biological data. For analysis and extracting meaningful information new fast methods are required. In the bioinformatics the raise of the amount of data is even more rapid than the other fields of the computer science. Despite the fact that for many of the bioinformatics problems effective algorithms are known, they do not work due to the size of the data.

The given problem we need to solve is approximate string matching for the input string.

The thesis is divided into six chapters with following structure:

In Chapter *Definitions* 1 we will look at the data structures that are commonly used in the bioinformatics. We will discuss their effective implementations, time and space complexity.

In Chapter *Bioinformatics* 2 the basic introduction into the the bilogy will be provided to understand used terminology.

In Chapter *Existing Solutions* 3 we will overview and compare currently used tools and applications to solve given problem.

In Chapter *Analysis and Design* 4 we will discuss our proposed solution and its properties.

In Chapter Implementation 5 we will look on some of the interesting parts of our implementation.

In Chapter Experiments 6 our implementation is test against real data.

In Conclusion the results are summarize and the possible future improvements are discussed.

Chapter 1

Definitions

This chapter defines data structures that can be used for indexing a text. We will discuss their construction complexity and operatins that are later used in an implementation. For more informations or examples feel free to see Prof. Crochemore publication [CHL07].

1.1 Basic Definitions

Definition 1.1 (Alphabet)
An *alphabet* Σ is a finite non-empty ordered set of *symbols.* We will refer to Σ as set of symbols `A`, `C`, `G`, `T`.

Definition 1.2 (String)
A *string* over Σ is any sequence of symbols from Σ.

Definition 1.3 (Size of a string)
The *size of a string* w is the number of symbols in the text $w \in \Sigma^*$ and is denoted $|w|$.

Definition 1.4 (Text)
The *text* $t = t_1 \ldots t_n$ is an input string that is indexed.

Definition 1.5 (Pattern)
The *pattern* $p = p_1 \ldots p_p$ is a string that we look for in the text.

Definition 1.6 (Substring)
A string x is a *substring* of a string y, if $y = uxv$, where x, y, u, $v \in \Sigma^*$.

Definition 1.7 (Subsequence)
A string $x = x_1 \ldots x_k$ is a *subsequence* of a string y, if $y = y_1 x_1 \ldots y_k x_k y_{k+1}$, where x, y, $y_1 \ldots y_{k+1}$, $x_1 \ldots x_k \in \Sigma^*$, $k \geq$.

Definition 1.8 (Prefix)
A *prefix* of a string $T = t_1 \dots t_n$ is a substring $\textit{pref}(m) = t_1 \dots t_m$, where $m \leq n$.

Definition 1.9 (Suffix)
A *suffix* of a string $T = t_1 \dots t_n$ is a substring $\textit{suff}(m) = t_{n-m+1} \dots t_n$, where $m \leq n$.

Definition 1.10 (K-mer)
A *k-mer u* is a substring of a string t of length k.

1.2 Self-Index

Self-index is a data compressed data structure that allow indexing and retrieving any substring of the text. Indexing allows us to search through data generally in time complexity linear to the size of the pattern p instead of linear in the size of the text t. We suppose that $|p| \ll |t|$.

Inverted Index is an index data structure storing mapping for every unique term to a collection of the *texts*. There are two types of *inverted index*. First is *inverted file index*, that stores only information about in which the *text* the *term* appears and second, *full inverted index*, that additionally stores information about exact locations of all *terms* within *texts*. Both data structures support quick answer to the number of the *texts* with the *term*. The *term* may be an English word, a number, or a k-mer of binary sequence or a DNA sequence. Look up time for any term is fast, it can be constant using hash tables or logarithmic using sorted array.

1.3 Suffix Array

A *suffix array* (*SA*) is a lexicographically sorted array of all suffixes of a given text T introduced by Manber and Myers in [MM90]. Suffix array itself contains pointers to the start of each suffix of the text. More formaly, for every $i, j, i < j$ holds that $\textit{suff}(SA[i]) < \textit{suff}(SA[j])$. Using suffix array we can search for the pattern p in the text t in $\mathcal{O}(|p| + \log(|t|))$ time.

Suffix array advantage is the size of an index opposed to other uncompressed indexing data structure. However, it may be still a lot of space to store the *SA*. The number of elements of the *SA* is the same as the size of the indexed text. So we need exactly $|t|$ integers to represent *SA* and also the maximum value of an integer is equal to the $|t|$. The most common integer sizes are 32 and 64 bits long. We can use 32 bits long integer only for texts long up to 2^{32} characters ($\sim$ 4GB file size of a plain text) and if the text is longer we need to use 64 bits per integer.

The DNA base pairs alphabet contains only 4 symbols, so for representation of the DNA we can use only 2 bits of memory for each symbol. However, for the *SA* we need to use numbers that can represent each position in the text. Meaning that the *SA*, constructed for the DNA text t, requires $|t| \cdot \textit{integer\_size}$ space. For 32 bit integers, space required to store the *SA*, is more than sixteen times greater than the size of the original text.

1.3.1 Construction of the Suffix Array

For the *SA* construction we need to have an ordered alphabet Σ with one additional symbol \$ with the lowest lexicographical value.

Before sorting suffixes, the \$ symbol is appended to the end of the *text* indicating its end.

Sort operation itself can be trivial but also ineffective. We can just compare two suffixes and place them in the correct order, but for large *texts* more effective algorithms were developed. Example 1.1 shows suffixes of the word *banana* and resulting suffix array.

Note that arrays and vectors are zero-based (indexes starts from 0) and string positions are one-based (positions starts from 1). Also the size of the *text* $|t|$ will be denoted as n.

Example 1.1 (All suffixes of the word *banana* and corresponding sorted array)

Position	Suffix
1	banana\$
2	anana\$
3	nana\$
4	ana\$
5	na\$
6	a\$
7	\$

$\xrightarrow{sort}$

Index	SA	Suffix
-1	7	\$
0	6	a\$
1	4	ana\$
2	2	anana\$
3	1	banana\$
4	5	na\$
5	3	nana\$

1.3.2 Time and Space Complexity

1.3.2.1 Construction complexity

The most naive algorithm takes $\mathcal{O}(n^2 \log n)$ time to construct *SA* by simply sorting all suffixes of the text t, which takes $\mathcal{O}(n \log n)$ comparisons, where each comparison takes $\mathcal{O}(n)$ time.

Time complexity of the *SA* construction is $\mathcal{O}(n)$ since 2003 by [BK03, KS03, Morb]. Although it is asymptoticaly optimal algorithm, it contains large hidden constant. Algorithm with construction complexity $\mathcal{O}(n \log n)$ are also still widely used. Most notable is [Mora].

Space required to store final the *SA* is $O(n \log(n))$ of memory (see the previous section). The algorithms are asymptoticaly always $\mathcal{O}(n)$ although an additional temporary memory is required to sort the suffixes. The difference is in the hidden constants.

Table 1.1: Summary of time & space complexity of SA operation

Operation	Time complexity	Space complexity
Construction	$O(n)$	$O(n \log(n))$
Search	$O(\|p\|)$	$O(1)$

Many existing algorithms can operate using only internal memory [Morb, Mora]. To reduce internal memory requirements, algorithms which use also external memory [KK14, KKP15, LTC13] were developed. For example algorithm [LTC13] use only about $1.5n$ of internal memory at any time of a construction of the *SA* but about $10n$ of external memory for temporary data. Memory requirements of an algorithm that use only internal memory depends mainly on the size of an integer that is used to represent the *SA* elements. Algorithm [Mora] uses $5n$ or $9n$ for 32 or 64 bits long integer, respectively.

The algorithm that sorts only within internal memory is generally faster than the algorithm that use also external memory (because it is rapidly slower to read data from external memory). However, they are extremely useful when we don't have enough internal memory to process long input.

1.3.2.2 Search operation complexity

Search over sorted suffix array can be performed in $O(|p|\log(n))$ time using binary search algorithm (comparing $|p|$ characters in total). This was improved to $O(|p| + \log(n))$ by [MM93] using additional *longest common prefix array* (*LCP array*) information ($\mathcal{O}(n)$ additional space) and even faster search achived by [AKO04] that needs only $\mathcal{O}(|p|)$ time using table called *child-table* (both *LCP array* and *child-table* can be stored in additional $\mathcal{O}(n)$ bytes, but is asymptoticaly still $\mathcal{O}(n \log n)$).

Table 1.1 summarizes time and space complexity of construction and search operation for best known algorithms today.

Comparison of construct algorithms can be found online here [sac, Morb, PST07] and in Chapter 4.

1.4 Burrows–Wheeler transform

The Burrows-Wheeler transformation was invented by Michael Burrows and David Wheeler [BW94] (in short *BWT*) can be defined as follows, given a string T of length n, sort all its n cyclic rotations in the same way as *suffix array* in Section 1.2. Sorting is exactly the same because comparision of a cyclic rotated strings terminates at the \$ symbol. All cyclic rotations of the *text* forms so called *burrows-wheeler matrix* (in short *BWM*) and *BWT* is the string constructed from the last column.

1	2	3	4	5	6	7
b	a	n	a	n	a	\$
a	n	a	n	a	\$	b
n	a	n	a	\$	b	a
a	n	a	\$	b	a	n
n	a	\$	b	a	n	a
a	\$	b	a	n	a	n
\$	b	a	n	a	n	a

1	2	3	4	5	6	7
\$	b	a	n	a	n	a
a	\$	b	a	n	a	n
a	n	a	\$	b	a	n
a	n	a	n	a	\$	b
b	a	n	a	n	a	\$
n	a	\$	b	a	n	a
n	a	n	a	\$	b	a

Table 1.2: Burrows-Wheeler tranformation using Burrows-Wheeler matrix for string *banana*

$$BWT[i] = \begin{cases} T_{SA[i]-1} & SA[i] > 1 \\ T_{|T|} & SA[i] == 1 \end{cases} \tag{1.1}$$

Figure 1.1: Connection between the SA and the BWT for the text T.

More formaly, the BWT can be construct using the *suffix array* and the original text as defined in Equation 1.1.

The BWT has become very popular for the *text* compression application, because of two important properties. First the BWT tends to have same symbols grouped together so it is more suitable to compress transformation using techniques as *move to front* or *run-length encoding.* In an Example 1.2 we can observe that after the BWT is applied to the *text* t resulting string tends to have same symbols grouped. However, this property depends on the input text. For highly repetitive strings there are longer sequences of similar symbols and so better compression ratio is be achieved.

Note that the symbol \$ is used only for sorting suffixes and resulting BWT may or may not include it. It works either way.

Example 1.2 (BWT [GS12])

T:	now␣is␣the␣time␣for␣the␣truly␣nice␣people␣to␣come␣to␣the␣party
BWT:	oewyeeeosreeeepi␣mhchlmhp␣tttnt␣puio␣yttcefn␣␣ooati␣␣␣␣␣␣rrolt

The second property is that BWT is inversible, allowing us to regenerate all rotations and so the original text. To make inverse operation possible, we need to store the index of a non-rotated original text in the SA. This index is presented as I. Without this information it would be imposible to guess where the *text* starts because all rotations have same resulted BWT.

1.4.1 Inverse of BWT

Reconstruction of the original text from the BWT can be done in several ways. The first method use reconstruction of the BWM by sorting and appending columns. It is important to realize that when we sort the BWT we get the first column of the BWM. We append it to the right of the BWT so two columns of the BWM are now reconstructed. Next $i+1$ column is reconstructed by sorting previous i columns, remember only last i^{th} column and append that column to the right of the not sorted i previous columns. The original text is at row I or where the \$ symbol is at the last position.

The second method is the last-to-front mapping (*LF mapping*). By applying the BWT on the *text*, all occurrences of each symbol in the BWT have the same relative order as in the original text. It is something similar to the stable property of sorting algorithms. This can also be formulated as Lemma 1.1.

Next, we need to create addtitional array C of length $|\Sigma|$, such that $C[c] =$ the number of occurrences of all symbols lexically smaller than c in the *text*, that is defined for all symbols from Σ and a function $Occ(c,i) =$ the number of occurrences of the symbol c in the prefix $BWT[0 \dots i]$. Then we can define a function $LF(i)$ as follows.

$$LF(i) = C[BWT[i]] + Occ(BWT[i], i) \tag{1.2}$$

Lemma 1.1 (LF mapping property)
The i^{th} occurrence of a symbol c in the BWT has the same rank as the i^{th} occurrence of c in the original text.

The previous equation 1.2 can be interpreted as that the symbol c at position i in the BWT is located in the *text* at position $LF(i)$. We say that the symbol c is symbol t_j from the original text at position j. Using the *LF mapping* we can get symbol t_{j-1} (symbol right before c in the original text) by $BWT[LF(i)]$. We can now reconstruct the original text by applying the *LF mapping*, starting at the last position of the original text (position I of BWT), n times or until the \$ symbol is reached.

1.5 FM-Index

The *FM-index* is a self-index created by Paolo Ferragina and Giovanni Manzini [FM00]. They describe how the BWT, together with additional data structure, can be used as a space-efficient index of the text t. The *FM-index* use the *LF mapping* to perform backward search to find both the number of occurrences of the pattern and their locations.

Example 1.3 is used to define and demonstrate operations on the *FM-index*. Indexed text is *banana*. The *FM-index* consists of the Burrows-Wheeler transformation of the text, table $C[c]$ and function $Occ(c,i)$ as defined in the previous section.

i	F	BWT
0	$	a
1	a	n
2	a	n
3	a	b
4	b	$
5	n	a
6	n	a

c	$	a	b	n
C[c]	0	1	4	5

Table 1.3: FM-index example: data structures

$$\begin{aligned} sp &= C[c] + Occ(c, sp - 1) + 1 \\ ep &= C[c] + Occ(c, ep) \end{aligned}$$

Figure 1.2: Equations for updating borders of search interval

1.5.1 Searching

We want to search for occurrences of a pattern $p = ana$ over the text t. The operation iterates backwards over the pattern. At the beginning of the search, variables $sp = 1$, $ep = |t|$ and $i = |p|$ are defined. Variables sp and ep (starting and ending position, respectively) are used as an interval where a suffix of the pattern is found in the BWT of the *text*.

Every step of the search operation starts by defining $c = p_i$, i^{th} symbol of the p. The sp and ep are updated by Equations 1.2 and i is decreased. At this point Lemma 1.1 was used. The sp and ep are boundary positions of the interval over BWT. Repeat the steps until either $i == 0$ or $sp > ep$ is reached.

Simple example: First proceed by finding the rows beginning with the last character of the pattern $p_{|p|}$, a in the example. Update $sp = C[a] + Occ(a, 0) + 1 = 2$ and $ep = C[a] + Occ(a, 3) = 4$ to get new sp and ep. i is decreased 1. At the end of every step sp and ep form boundary positions of the $suff(i)$ of the p. The number of steps is $|p|$.

In short, we apply the *LF mapping* repeatedly to find the range of rows that starts with the $suff(i)$ of the p until $suff(0)$ or the range becomes empty, in which case p does not occur in t.

For more detailed example you can visit [Bow11a].

1.5.1.1 Time complexity

Lookup to the table C needs constant time, the function $Occ(c, i)$ can be performed in constant time by remembering the number of the occurrences for every position and every symbols of the alphabet. Time complexity of the funtion Occ is also $\mathcal{O}(1)$. These operations are done for every character of the pattern p with length $|p|$.

Time complexity of search is $\mathcal{O}(|p|)$. We need to mention here that constant lookup time for the function *Occ* takes too much space and it is not suitable for DNA data or long texts in general. This means that time complexity of search operation in our case will be defined as $O(|p|\gamma)$, γ is time complexity for the function *Occ* and will be discussed at Section 1.7.2.

1.5.2 Locating

When we find all occurrences of the pattern p we can easily determine their positions in the *text* from the *SA*. We already have range of the indexes to the *BWT* for all occurrences and by just looking at the same indexes to the *SA* we will get the position within the text for each index.

1.5.2.1 Time complexity

For locating we use search operation ($\mathcal{O}(|p|\gamma)$ time) and then determine positions from the *SA*. This operation also takes constant time but we need too much space to sacrifice for it.

Time complexity is therefore $\mathcal{O}(\mathit{search} + \mathit{location}) = \mathcal{O}(|p|\gamma)$.

1.5.3 Extracting

Extract operation using complete *BWT* is trivial. We just use *LF mapping* from any position of the *BWT* and we will get string as long as we need. However, later representation of the *BWT* will need slightly different approach.

Time complexity is $\mathcal{O}(\mathit{search}) + \mathcal{O}(l\tau)$, where l is a length of the extracted string and τ is time complexity of the *LF mapping* operation ($\mathcal{O}(1)$).

Simplified as $\mathcal{O}(|p|\gamma + l)$.

SA is memory consuming. The bright side we don't have to keep the whole *SA* but just some of its elements. This is so called *Compressed Suffix Array* (*CSA*) which we will discussed lates at Section 1.6.

1.6 Compressed Suffix Array

1.6.1 Ψ based compression

Space requiremens of the suffix array led to creating a concept of *compressed suffix array* (*CSA*). Introduced by [GV00] with motivation to create space-efficient text indexing method that support fast string search. Their representation of the *CSA* is a hierarchical data structure composed of the levels $k \geq 0$. The "zero" level ($k = 0$) is the original *SA*. The first level ($k = 1$) stores half of the original information, $\frac{n}{2}$ elements. The k^{th} level stores exactly $\frac{n}{2^k}$ elements of the *SA*. As we can observe, at every level $k + 1$ we have to store half of the elements of the level k. Authors recommend to use level l equal to $\lceil \log \log n \rceil$.

Construction of a level SA_{k+1} contains four main steps to transform level SA_k into an equivalent but more succinct representation:

Step 1 Produce a bit vector B_k of n_k bits, such that $B_k[i] = 1$ if $SA_k[i]$ is even and $B_k[i] = 0$ if $SA_k[i]$ is odd.

Step 2 Map each 0 in B_k onto its companion 1. (We say that a certain 0 is the companion of a certain 1 if the odd entry in SA associated with the 0 is one less than the entry in SA associated with the $B_k = 1$.) We can define for this Ψ function as follows:

$$\Psi_k(i) = \begin{cases} j & \text{if } SA_k[i] \text{ is odd and } SA_k[j] = SA_k[i] + 1; \\ i & \text{otherwise.} \end{cases}$$

Step 3 Compute the number of 1s for every prefix of B_k.

Step 4 Create new SA_{k+1} with elements from SA_k that have 1s in B_k. Divide each of them by 2.

Succinct representation contains only SA_l, B_l and function Ψ. The previous levels are discarded.

When we want to reconstruct element from original SA at index i we use Algorithm 1 for CSA with level l. Example of construction of a CSA where level $l = 3$ is in Tables 1.4.

Algorithm 1 Recursive lookup of entry $SA_k[i]$ in a CSA. Variable l is level of the CSA.

```
procedure RLOOKUP(i, k)
    if $k = l$ then return $SA_l[i]$
    elsereturn $2 \cdot \text{RLOOKUP}(rank_k(\Psi_k(i)), k+1) + (B_k[i] - 1)$
    end if
end procedure
```

Time complexity is $\mathcal{O}(\log\log n)$ when we use level $l = \lceil \log\log n \rceil$. Time complexity therefore depends on level l. Other operations are considered to be constant time.

1.6.2 Sampling based compression

Another way to reduce space requirements is to simply sample the SA. Sampling takes every k^{th} position of the SA, discarding others and re-creating them as needed. Example 1.5 shows sampled CSA with sample rate equals to 3.

As opposed to the Ψ based CSA, sampled CSA needs the BWT with the array C to be able to reconstruct missing elements.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SA_1	8	14	5	2	12	16	7	15	6	9	3	10	13	4	1	11
B_1	1	1	0	1	1	1	0	0	1	0	0	1	0	1	0	0
$rank_1$	1	2	2	3	4	5	5	5	6	6	6	7	7	8	8	8
Ψ_1	1	2	9	4	5	6	1	6	9	12	14	12	2	14	4	5

	1	2	3	4	5	6	7	8
SA_2	4	7	1	6	8	3	5	2
B_2	1	0	0	1	1	0	0	1
$rank_2$	1	1	1	2	3	3	3	4
Ψ_2	1	5	8	4	5	1	4	9

	1	2	3	4
SA_3	2	3	4	1

Table 1.4: Compressed Suffix Array using Ψ function

If we want to look up $SA[i]$ but it has not been sampled, we can use *LF mapping* $LF(i)$ to step one position to the left in the text and to index j in the BWT and look if element $CSA[j]$ is sampled, if not, we apply *LF mapping* until reaching sampled position. To reconstruct the position from the $CSA[i]$ we simply add the number of steps that were used to reach $CSA[j]$ from $CSA[i]$ to the position from $CSA[j]$. $CSA[i] = \text{steps} + CSA[j]$.

i	F	BWT
1	$	a
2	a	n
3	a	n
4	a	b
5	b	$
6	n	a
7	n	a

i	F	BWT	CSA
1	$	a	7
2	a	n	-
3	a	n	-
4	a	b	2
5	b	$	-
6	n	a	-
7	n	a	3

Table 1.5: Compressed Suffix Array using sampling

Time to recalculate any position of the SA from sampled CSA is equal to the number of *LF mapping* steps we have to use. The maximum number of steps is limited by constant k, so in the worst case we need to apply k-times *LF mapping* to get to the sampled element. If we choose k as small constant, we can consider operation to perform in $\mathcal{O}(1)$ time.

1.7 Succinct Data Structures

According to [Mun], a succinct data structure is a data structure which uses an amount of space of representation close to the information theoretic lower bound, but is still time-efficient for query operations. The goal is to perform in the same time as the “normal” data structure without such space constraints.

The concept was originally introduced by Jacobson [Jac88] on static graphs, trees and bit vectors. Defined two functions *rank* and *select*. Unlike lossless data compression algorithms, succinct representation do not need data to be decompressed before performing operations on them.

1.7.1 Rank and Select

Rank & select data structures are one of the fundamental building blocks for many modern succinct data structures. Asymptotic space requirement for these two data structures is same as asymptotic lower bound of information theory.

1.7.1.1 Rank

Data structure *rank* with defined function $rank(c, i)$ is constructed over the text t returns the number of occurrences of the symbol c in the prefix $pref(i)$ of the *text*. The function *Occ* defined in the same way was used in the *LF mapping* and it can be replaced by the *rank*. Example 1.3.

Example 1.3 (Example of the rank operation)

	$\downarrow$									
i	0	1	2	3	4	5	6	7	8	9
	0	1	1	0	1	0	1	0	0	1

$$rank(0) = 0$$

						$\downarrow$				
i	0	1	2	3	4	5	6	7	8	9
	0	1	1	0	1	0	1	0	0	1

$$rank(5) = 3$$

The *Occ* function can be performed without any additional space, but at the cost of the query time ($\mathcal{O}(|t|)$). A constant time query is achieved by storing the number of occurences for each position of the *text*, but it uses too much space.

A trivial representation of the *rank* is to store precomputed every k^{th} position of the array. Query $rank(i)$ to a position i, which is precomputed, is performed in constant time by simple access to the table. Query to any position between two precomputed, j, $i < j < i + k$, need to scan the text from the position i to the position j using linear algorithm, find the number of occurrences within the interval and add it to $rank(i)$ value.

If we choose the k small enough time complexity of the *rank* query is assumed to be $\mathcal{O}(1)$. Besides original text we need additional table K of precomputed elements. Table K has $l = \frac{|t|}{k}$ elements and every element needs to be big enough to store the largest number of occurrences of the symbol over entire text. Let say x is the largest number of occurrences than each element needs $\lceil \log x \rceil$ bits of space. Overall space required $|t| + l\lceil \log x \rceil |\Sigma|$.

Less trivial and more interesting representation of the *rank* data structure performs query operation also in $\mathcal{O}(1)$ time, but space requirements are slightly better. An idea is similar to the *range minimum query.* For simplicity we will define it only for bit vectors with function *rank*(i) defined as the number of occurrences of the bit 1 in the prefix *pref*(i) of the text $t \in \{0,1\}^*$.

Divide an array into the *blocks* of length $b = \lfloor \frac{\log n}{2} \rfloor$. Consecutive *blocks* are grouped into the *superblocks* of length $s = b\lfloor \log n \rfloor$. For each *superblock* j, $0 \leq j \leq \lfloor \frac{|t|}{s} \rfloor$ we define a table $R_s[j] = \mathit{rank}(j \cdot s)$, which is the number of occurrences of the 1 within $\mathit{pref}(j \cdot s)$. Next, for every *block* k of the *superblock* j, $0 \leq k \leq \lfloor \frac{|t|}{b} \rfloor$, we define a table R_b as follows $R_b[k] = \mathit{rank}(k \cdot b) - \mathit{rank}(j \cdot s)$, number of 1s from start of the *superblock* j to the position $k \cdot b$.

We have one table R_s and for every *superblock* one table R_b. The table R_s needs $\mathcal{O}(\frac{|t|}{\log |t|})$ bits and the R_b $\mathcal{O}(\frac{|t| \log\log |t|}{\log |t|})$ bits. Asymptoticaly it has same space requirements as previous representation, but may be better on real data. We can find different definitions with slightly different space requirements, for example in [GGMN05].

1.7.1.2 Select

The *Select* is an inverse operation to the *rank.* The function *select*(c, i) answers the question at which position is i^{th} occurrence of the symbol c. Example 1.4.

Example 1.4 (Example of the select operation)

i	0	1	2	3	4	5	6	7	8	9
	0	1	1	0	1	0	1	0	0	1
	↑									

$$\mathit{select}(0) = 0$$

i	0	1	2	3	4	5	6	7	8	9
	0	1	1	0	1	0	1	0	0	1
										↑

$$\mathit{select}(5) = 9$$

A simple solution in $\mathcal{O}(\log n)$ time only requires to use *rank* data structure. If we want a j^{th} position of the symbol c, *select*(c, j), we use *binary search* over *rank* structure to find $\mathit{rank}(c, i) = j$ and $\mathit{rank}(c, i-1) = j - 1$.

The *rank* query is constant, but we need to use *binary search* with time complexity $\mathcal{O}(\log n)$. Overall time complexity is $\mathcal{O}(\log n)$ without any additional space.

Constant time version of the *select* is more complex than the *rank* solution. Some of the solutions, like [Cla98], uses too much space overhead to enable constant time *select* query to be practical.

In the next section we will overview some of the most used representations for the bit vectors that supports both *rank* and *select* queries in constant time.

1.7.1.3 RRR

The *RRR* data structure, introduced by [RRR02], supports both *rank* and *select* queries in $\mathcal{O}(1)$ time and requires $\mathcal{B}(n, m) + o(n) + \mathcal{O}(\log \log m)$ bits to store the text of size n, where $\mathcal{B}(n, m) = \lceil \log \binom{m}{n} \rceil$ is the minimum number of bits required to store any n-element subset from a universe of size m.

The construction of the *RRR* structure is similar to the previous one. We divide bit vector into blocks of the size b. Blocks are grouped into superblocks of size f. This already allows us to construct an index to enable $\mathcal{O}(1)$ rank query. In the Example 1.5 the first step is shown for block size $b = 5$ and superblock of three blocks, $f = 3$.

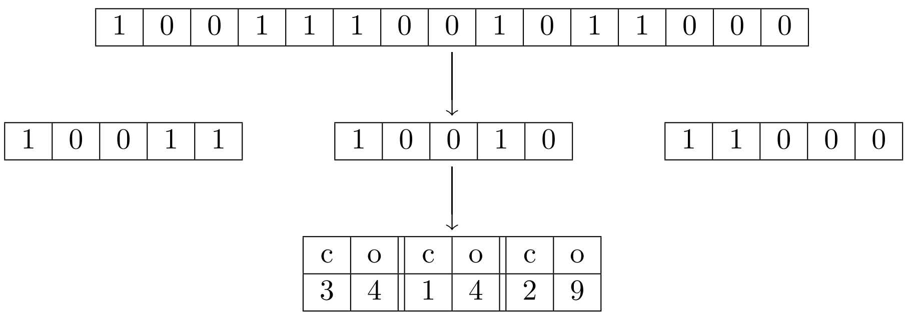
Example 1.5 (*RRR* construction)

Superblocks		s_0			s_1			s_2	
Blocks	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
	00101	00110	10110	10000	01001	01001	10110	00101	10110

As the first step, we replace each block with a pair of values. A *class* value C and a *offset* value O. We use C with O together as a look up key into the table of precomputed *ranks*. For every block i, $0 \leq i < \lfloor \frac{|t|}{b} \rfloor$, we define C to be $C(i) = popcount(i)$, where function *popcount*() returns the number of set bits in the block i. Each block is now in exactly one *class* c, $0 \leq c < b$.

Example 1.6 show this process. Bit vector is divided into three blocks and converted to pair representation of a *class* and an *offset*.

Example 1.6 (*RRR* construction)

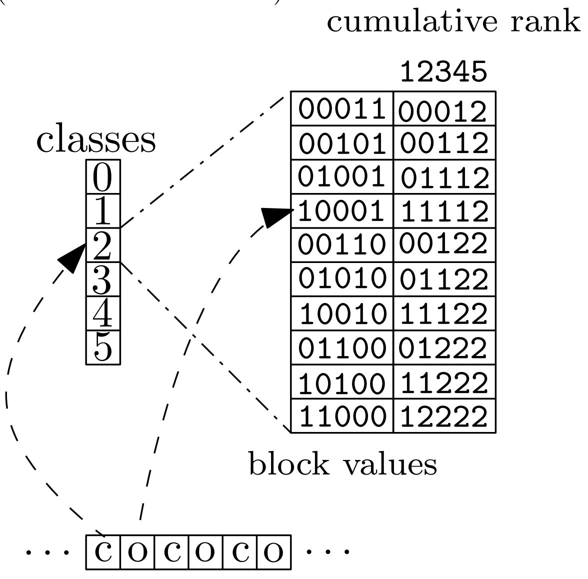


The *class* number is used as a direct index to the table G of the size b. For the *class* c, $0 \leq c < b$, G contains a table at $G[c]$ with $\binom{b}{c}$ elements. Table $G[c]$ corresponds to all possible permutations of 1s bits within a single block which have *popcount* equal to c. Example 1.7 shows all possible permutations for the block b_i of the size $b = 5$ and $popcount(b) = 2$.

Example 1.7 (All permutations for block of size 5 and $popcount = 2$.)

block	*class*
00011	2
00101	2
01001	2
10001	2
00110	2
01010	2
10010	2
01100	2
10100	2
11000	2

Example 1.8 (*RRR* data structures)



For every element of the top level of table G we need $\lceil \log b \rceil$ bits, overall $b * \lceil \log b \rceil$, but asymptoticaly $\mathcal{O}(\log b)$. *Offset* values vary in the size. Every *class* c needs asymptoticaly $\mathcal{O}(\log \binom{b}{c})$ bits on element of $G[c]$. The number of elements is $\binom{b}{c}$.

We still can't perform *rank* query in the constant time, but at least we have achieved a possible compression, because we can use variable length code to store *offset* O values. Thanks to C value we know how many bits follow for paired O value.

To be able to perform asymptoticaly constant time query we use a method discussed by [Mun96]. For every superblock with start boundary i we store the *rank* over the *pref*(i) (from the start up to the superblock boundary i).

method	**size** (bits)	**rank**	**select**
esp	$nH_0(S) + o(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
recrank	$1.44m \log \frac{n}{m} + m + o(n)$	$\mathcal{O}(\log \frac{n}{m})$	$\mathcal{O}(\log \frac{n}{m})$
vcode	$m \log \frac{n}{\log^2 n} + o(n)$	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(\log n)$
sarray	$m \log \frac{n}{m} + 2m + o(m)$	$\mathcal{O}(\log \frac{n}{m}) + \mathcal{O}(\log^4 \frac{m}{\log n})$	$\mathcal{O}(\log^4 \frac{m}{\log n})$
darray	$n + o(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^4 \frac{m}{\log n})$

Table 1.6: The space and time results for *esp*, *recrank*, *vcode*, *sarray* and *darray*

Additionally, we store a prefix sum of the bits, this allows us to iterate only over the blocks within the required superblock. Now we can perform asymptoticaly constant time *rank* query.

To calculate *rank*(i) we use procedure as defined in [Bow11b]:

Step 1 We calculate in which block our index is as $i_b = \frac{i}{b}$.

Step 2 Calculate which superblock our block resides in as $i_s = \frac{i_b}{f}$, f is a number of the blocks within one superblock.

Step 3 Use pre-calculated sum of previous *ranks* up to the superblock i_s boundary and add it to the result.

Step 4 For every block before i_b, we add its *class* value c to the result. It is equal to the *rank* query over entire block.

Step 5 We perform *rank*(j) query over block i_b from block start to the position $j = i \mod b$.

Only non-constant operation is iteration over the *blocks* within one *superblock*. The number of blocks is bound to the constant f. Other values are pre-computed. We have $\mathcal{O}(1)$ time *rank* query with reasonable additional space requirements.

For the *select*(i) query, it can be answered in $\mathcal{O}(1)$ time, it will need additonal space. Practical implementation by [CN09] uses *binary search* over pre-calculated prefix sum of the bits to find superblock i_s, *prefix sum* $< i$. Then iterate over the blocks within the *superblock* until we found block where i^{th} symbol is.

In the worst case, time complexity is $\mathcal{O}(\log \frac{|t|}{f})$ and $\mathcal{O}(1)$ on average.

1.7.1.4 SD Array

Another *rank* and *select* representations were introduced by [OS06]. Representations are summarized in the Table 1.6 for an ordered set $S \subset \{0, 1, \ldots, n-1\}$ with m elements. $H_0(S) \leq 1$ is the zeroth order empirical entropy of S.

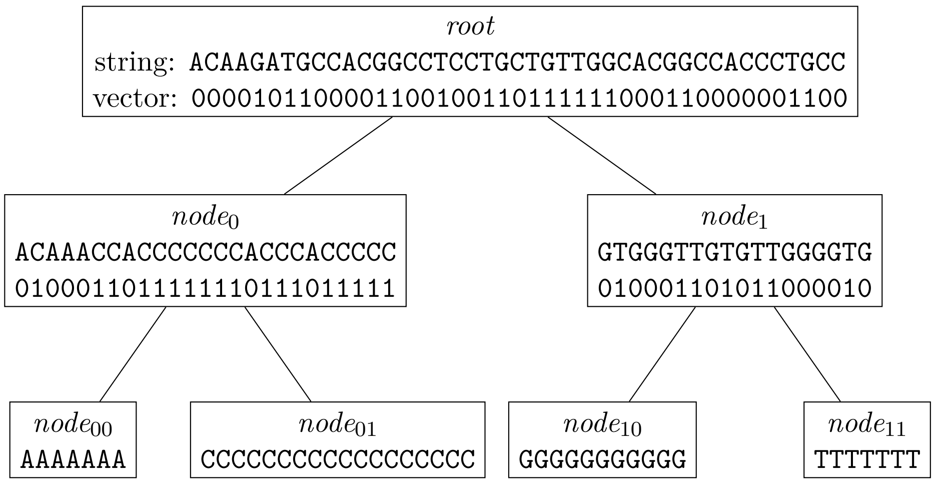


Figure 1.3: Wavelet tree of a string

Notable representation is so called *SDarray*. The idea is to use two different techniques for *sparse* and *dense* vectors. *Sarray* is used for sparse vectors. For an vector B of n elements and m ones bits (set bits, 1s), we define table $x[0 \ldots m-1]$ such that $x[i] = \textit{select}(i+1)$, $0 \leq i < m$, over vector B.

1.7.2 Wavelet Tree

The *wavelet tree* is a data structure representing a string using a hierarchy of a bit vectors. The data structure were introduced by [GGV03]. It generalizes the *rank* and *select* operations for better representation of an arbitrary alphabet.

It the original paper it was used to represent *suffix array*, but it is usable for any *string* over any *alphabet* Σ^0. *Wavelet tree* converts a string into a balanced binary tree of a bits vectors. Recursively divides an alphabet Σ^i into two subsets, Σ^i_l and Σ^i_r, at each level until leaf subsets contains only one symbol c from original alphabet. Every node of the tree represents a *subsequence* t^j of original the *text* t, contains symbols from particular subset Σ^j. Bit 0 replace symbols from the Σ^j_l and bit 1 replace symbols from the Σ^j_r.

Figure 1.3 of a preparation and construction of an *wavelet tree* of a string `ACAAGATGCCACGGCCTCCTGCTGTTGGCACGGCCACCCTGCC`.

1 We start with alphabet $\Sigma^0 = A, C, G, T$ and a string $t^0 =$ `ACAAGATGCC ACGGCCTCCTGCTGTTGGCACGGCCACCCTGCC`.

2 We divide Σ into two subsets $\Sigma_l = A, C$ and $\Sigma_r = G, T$.

3 We create bit vector b^0 by replacing all symbols $\in \Sigma_l$ with 0 bit and symbols from Σ_r with 1 bit.

4 Root node $b^0 =$ `000010110000110010011011111110001100000011 00`, $0 \in \Sigma_l$, $1 \in \Sigma_r$.

5 We create a subsequence of the t^0 where $b^0[i] = 0$ as t_l^1 and $t_r^1 = t^0[j], \forall j \ that \ b^0[j] = 1$.

6 $t_l^1 =$ `ACAAACCACCCCCCCACCCACCCCC` with alphabet $\Sigma^1 = A, C$, $t_r^1 =$ `GTGGGTTGTGTTGGGGTG` with $\Sigma^1 = G, T$.

7 For t_l^1 we divide Σ^1 into $\Sigma_l = A$ and $\Sigma_r = C$ and repeat previous steps for t^0.

8 We repeat these steps for all subsequences until alphabet of a node contains only one symbol.

We do not store substring for all nodes and also bit vectors of the leafs nodes. All bit vectors are represented as *RRR* data structures for fast *rank* and *select* queries.

Rank query is super simple. If we want $rank(A, 30)$, we start from the *root* node. As we know that A is in the Σ_l, we need to get the number of 0s in the text up to 30^{th} position. We use $Rank(30)$ to get the number of 1s bits. The number of 0 is a complement to the 30, $occurrences_0 = 30 - rank(30) = 16$. At the next level, $node_0$, to find occurrences of 0s up to the 16^{th}, $occurrences_1 = 16 - rank(16) = 6$. As we are at the last stored node, result of the $rank(A, 30) = 6$.

Asymptotic time complexity of the *rank* is $\mathcal{O}(\log \Sigma)$ if we use balanced binary tree.

Select query starts from the leaf. For example $select(A, 6)$, at the node $node_0$ finds number of all bits up to 6^{th} 0 bit, 16 bits. At the *root* node, we simply finds position of the 16^{th} 0 which is the result, $select(A, 6) = 30$.

Asymptotic time complexity, if we consider *select* over the *RRR* structure as $\mathcal{O}(1)$, is the same as the *rank* query $\mathcal{O}(\log \Sigma)$.

1.8 De Bruijn Graph

The *De Bruijn graph* is a directed graph, where each node represents a k-mer. An edge from a node u to a node v only exist if they overlap with $k-1$ symbols. A node u is denoted by $(u_1, \ldots, u_k)$ where $u_1, \ldots, u_k \in \Sigma$. For any pair of nodes $u = (u_1, \ldots, u_k)$ and $v = (v_1, \ldots, v_k)$ such that $u_2 = v_1, \ldots, u_k = v_{k-1}$, the graph has a directed edge from u to v labeled with v_k symbol.

We will consider the *De Bruijn graph* as a subgraph of complete *De Bruijn graph* that contains only nodes of k-mers that occurs in the text. Example 1.4 shows *De Bruijn graph* of a string `ACGCGCGCTACGTTACGT`.

As we can see, k-mers `ACG`, `CGC`, `GCG`, `TAC` and `CGT` occurs more than once in the text, but in the *De Bruijn graph* we use only one node to represent each

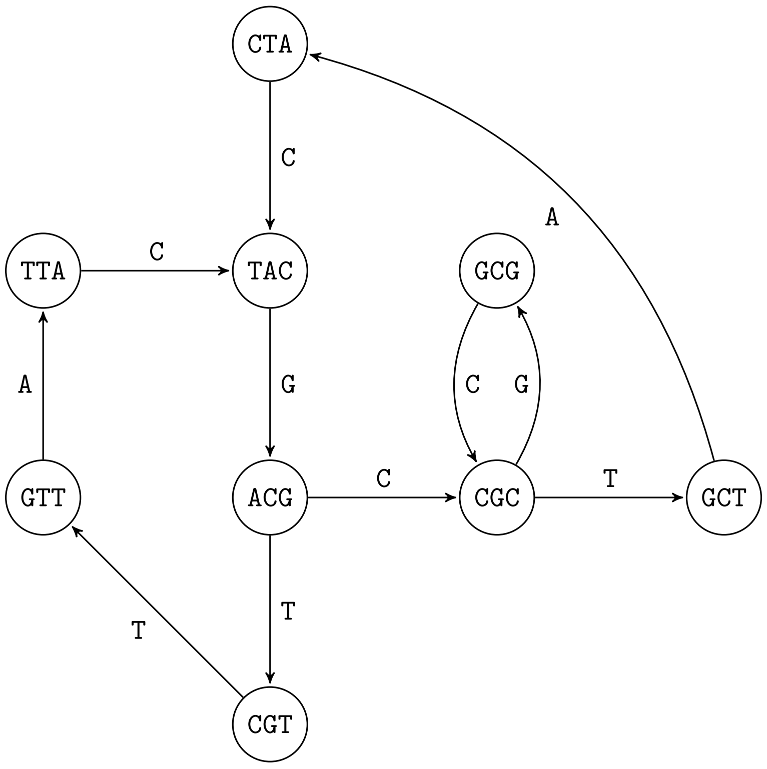


Figure 1.4: De Bruijn graph

k-mer. The maximal number of the output edges for each node is size of the alphabet $|\Sigma|$.

The maximal number of nodes is $|\Sigma|^k$. Every node represents k length subsequence of the text. As we traverse the graph, the last k edges define the current node.

In the bioinformatics De Bruijn graph is used in *DNA assembly* technique *De novo* [LFB$^+$14].

1.8.1 Succinct De Bruijn Graph

Besides other succinct or compressed representations of the *De Bruijn graph* [CB11, CR13], we will discuss representation of a succinct De Bruijn graph introduced by [BOSS12].

Let G be k-dimentional De Bruijn graph of a string t on alphabet Σ, n be the number of nodes and m be the number of edges. By definition succinct representation of G supports the following operations:

- *Forward*(i) return index of the last edge of the node pointed to by edge i.
- *Backward*(i) return index of the first edge that points to the node that the edge at i exits.
- *Outdegree*(v) returns the number of outgoing edges from the node v.
- *Outgoing*(v, c) returns the node w pointed to by the outgoing edge of the node v with the edge label c. If no such node exist returns -1.
- *Indegree*(v) return the number of incoming edges to the node v.
- *Incoming*(v, c) return the node $w = (w_1, \dots, w_k)$ such that there is an edge from w to the node v and $w_1 = c$. If no such node exists return -1.
- *Index*(p) return the index i of the node whose label is the string p of length k.

1.8.1.1 Construction

For constructon, lets consider that the *De Bruijn graph* is represented as an array H of pairs. Each pair consists of a *node* and an *edge* label. If a node i has more edges J, we will have $|J|$ pairs for the node i and $\forall j \in J$. An edge label of a pair l is denoted as *Edge*$[l]$ and a node label as *Node*$[l]$. Also we add k dummy symbols \$ at the start of the text. Nodes are sorted in the lexicographic order of reversals of associated node labels. Figure 1.5 shows an example.

We also need to define a set Σ^- of the size $|\Sigma|$, $\Sigma^- \cap \Sigma = \emptyset$. For each symbol $c \in \Sigma$ we define $c^- \in \Sigma^-$ and a function $\sigma(c^-) = c$. We use symbols from Σ^- as an edge symbols for some nodes instead of symbols from Σ. If any of two nodes i and j have the same suffix *suff*$(k-1)$ and the same edge label c, we exchange the edge label c in the second node with corresponding symbol $c^- \in \Sigma^-$, $\sigma(c^-) = c$. In other words, for every two nodes that can get to the node j using the same edge label c, we want to have only one edge symbol c in the edge array to maintain the Lemma 1.1.

The succinct representation consists of a three arrays. Let m be the number of pairs in the H. An array W of the size m, an array *last* of the size m over the binary alphabet $0, 1$ and an array F of the size $|\Sigma|$.

The array $W[i] = \textit{Edge}[i]$, $0 \leq i < m$, represents the labels of the edges. The *last* array is defined as follows: $\textit{last}[j] = 1$ if $j = m$ or the label of the node j is different from label of the node $j-1$, $\textit{Node}[j] \neq \textit{Node}[j-1]$ and $\textit{last}[j] = 0$ otherwise. The array F represents cumulative occurrences of the last symbols of node labels $\textit{Node}[j]_k$, defined exactly like the C array from the Section 1.4.1.

c	F[c]
\$	0
A	1
C	3
G	7
T	10

index	Last	Node			Edge
0	1	\$	\$	\$	T
1	1	C	G	A	C
2	1	\$	T	A	C
3	0	G	A	C	G
4	1	G	A	C	T
5	1	T	A	C	G^-
6	1	G	T	C	G
7	0	A	C	G	A
8	1	A	C	G	T
9	1	T	C	G	A^-
10	1	\$	\$	T	A
11	1	A	C	T	\$
12	1	C	G	T	C

Figure 1.5: *De Bruijn graph* representation.

The array W is represented using the *wavelet tree*, the bit vector *last* as *RRR* data structure and the F as simple table with constant access time.

In the next section we will discuss only operations *forward* and *index*. Remaining operations are defined in [BOSS12].

1.8.1.2 Forward

The last symbol of the node labels maintain the same relative order as the edge labels. Following an edge from the node i is simply finding the corresponding relatively positioned node j. For that the *rank* and *select* queries over arrays W and *last* are used.

First we access $W[i]$ to get the edge label c, then calculate $rank(c, i)$ to get the number r of occurrences of the symbol c up to the position i. Now to find r^{th} occurrence of the symbol c in the nodes, we take position j of first occurrence from the table F and select r^{th} occurrence using the *last* array. This require us to count how many 1s are before j, $occ = rank(c, j)$ and then select $(occ + r)^{th}$ occurrence, $forward(i) = select(c, occ + r)$.

$$
\begin{aligned}
&sp = pred(last, Outgoing(succ(last, sp), c_{d+1})) + 1 \\
&ep = Outgoing(ep, c_{d+1})
\end{aligned}
$$

Figure 1.6: Equations for updating borders of search interval, *De Bruijn graph*

1.8.1.3 Index

The algorithm is similar to the *FM index* search defined in Section 1.5.1. Only difference is Equations 1.6 for updating the *sp* and *ep* interval boundaries.

The function $pred(c, i) = select(c, rank(c, i))$ returns the position of the first occurrence of the symbol c when we scan the text from the position i to the start and the function $succ(c, i) = select(c, rank(c, i - 1) + 1)$ returns the position of the first occurrence of the symbol c when we scan the text from the position i to the end.

Chapter 2

Bioinformatics

This chapter provides brief introduction into the biology needed orient in the terminology. The bioinformatics field is focused on developing methods and algorithms to understand biological data.

2.1 DNA

DNA is a double helix structure consisting of four types of the molecules called *nucleotides*. The DNA is a long sequence of nucleotides of about 3 billion pairs. The nucleotide bases are *Adenine*, *Cytosine*, *Guanine* and *Thymine*. We can define them to form a four-letter alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

The pairs are two nucleotides attached together. Base `A` pairs only with `T` and `C` pairs only with `G`. We can say that `A` and T are complementary and the same holds for `C` and `G`. DNA have two strands of the complementary sequences. We can view the strands as two strings over the alphabet Σ. Given one strand we can construct the other strand by complementing its the bases. So we can only represented DNA by one strand to reduce the size.

DNA is wrapped into the structure called *chromosomes*. The number of *chromosomes* depends on specie. Human DNA is made from 23 *chromosomes* and all *chromosomes* form together the *genome*.

2.2 RNA

RNA is important molecule that is used to transfer genetic information. RNA has only one strand and the same four-letter alphabet with one change. The nucleotide `T` is replaced by *uracil* (`U`).

RNA has different types and these types are called by their function. For example *mRNA* is a *messenger RNA* that transmits the information from the DNA to the reproduction.

symbol	corresponding nucleotides
`A`	`A`
`C`	`C`
`G`	`G`
`T`	`T`
`U`	`U`
`R`	`A` or `G`
`Y`	`C`, `T` or `U`
`K`	`G`, `T` or `U`
`M`	`A` or `C`
`S`	`C` or `G`
`W`	`A`, `T` or `U`
`B`	`C`, `G`, `T` or `U`
`D`	`A`, `G`, `T` or `U`
`H`	`A`, `C`, `T` or `U`
`V`	`A`, `C` or `G`
`N`	`A C G T U`
`-`	gap

Table 2.1: Defined symbols of FASTA format.

Before we can process DNA sequences in computers, it needs to be digitized. This process is called DNA *sequencing*. While sequencing reading errors may occurs. Instead of guessing values these regions are represented by "dont know" symbols. Another type of error is produced by organisms itself by mutation. Mutation of single nucleotides is called *single nucleotide polymorphism* (*SNP*).

To describe the sequence of nucleotides we will use the *nt* prefix. For example, 20-nt string is a sequence of 20 nucleotides.

2.3 FASTA

A sequence in FASTA format begins with a single-line description, followed by lines of sequenced data. To represent one symbol from DNA is use 1 B for each symbol. Besides five nucleotides symbols, FASTA support another 12 symbols that represent different set of nucleotides. Table 2.1 shows all defined symbols with their corresponding set.

The description line always starts with `>` symbol.

Some of the other formats are using binary representation of the DNA, where each nucleotide (without `U`) can be represented by only 2 bit of memory. Binary formats are harder to process, but uses significantly less space.

2.4 CRISPR

The *CRISPR/Cas9* system is a genome engineering approaches, that supports genome editing. The systems consists of two components, a *guide RNA* (*gRNA*) and a *Cas9* enzyme. The *gRNA* is a short synthetic RNA sequence necessary for *Cas9* binding of a 20-nt user defined sequence which defines the genomic target to be modified.

The *Cas9* is used as a carrier that locates specific regions. These regions are defined by the *gRNA* followed by the *PAM* sequence. We can change the genomic target of *Cas9* by changing the target sequence *gRNA*.

PAM sequence has lenght of 3 to 6 nucleotides and is defined in FASTA format. For example PAM sequence `NGG` represents all possible strings of the length 3 where `N` is expanded by all possible symbols it represent.

To be more technical, the *gRNA*, *Cas9* and *PAM* are strings over the DNA alphabet and targeting their location in the genome is equal to searching operation. Therefore, *CRISPR/Cas9* system is a string matching problem.

A *guide* is a target region in the genome we want to edit. Due to the *SNP* we need to be able to search also for the sequences that are similar to the *guide* but they have one or more *SNP*. All sequences that have m or less *SNPs* from the *guide* are called *off-targets*.

The *CRISPR/Cas9* system can be used for various purposes. It can cut or enable targeted regions. For the cut operation, we can use single or double strand *gRNA* targeting sequences. Double strand targeting is more effective.

Using a *gRNA* to target specific region, we need to be able to *score* how likely it is to succeed. This is why we need the *off-targets*. The input *guide* is scored based on the occurrences of its *off-targets*. The more occurrences the lower chance that the *guide* succeed.

To actualy score single *guide* all of its *off-targets* needs to be scored first. Single *off-target* is scored based on the number of occurrences in the genome and the number of *SNPs*.

In the thesis all sequences are represented by FASTA format.

Chapter 3

Existing Solutions

All *CRISPR/Cas9* design systems are using the same main process. The input is a sequence or multiple sequences. If the input is multiple sequences each of them is considered as individual case and is processed separately.

First, all applications try to find potential guide RNA sequences in the input sequence. The input sequence is scanned for the PAM sequence, the guide is extracted as region before PAM sequence. For every possible guide, we scan entire genome to find the all off-targets with possible *SNP* mismatches. After the off-targets for each guides are found, the application scores guides and off-targets. Most applications uses scoring as defined in [HSW$^+$13].

A lot of tools that exists now for CRISPR design are available only online through web interface. For the tools that do not haves public information about implementation or source code, we will write something about their features.

Summary of the process of designing CRISPR guides:

1. Scan input sequence for all possible *guides* and their *PAM sequences.*
2. For every *guide* x find all occurrences of the x with maximum k mismatches. The sequences are called the *off-targets.*
3. Score every *off-target.*
4. Score every *guide.*
5. Show results.

3.1 Existing Libraries

3.1.1 Bowtie

The *Bowtie* introduced by [LTPS09, LTPS] is a software library that is widely used in the bioinformatics for sequence alignment an sequence analysis. The

Bowtie indexes the genome with the FM Index.

Thanks to its ease of integration and speed, many applications for CRISPR Design use the *Bowtie* to perform scans of the genome to find the *off-targets*. The design of the *Bowtie* search algorithm use some heuristics and thus is not that good for exhaustive search in the entire genome and may miss many potential *off-targets*.

3.2 Existing Applications

3.2.1 ZiFiT

Author	Zinc Finger Consortium [SMR+10]
Web page	`http://zifit.partners.org/ZiFiT/`
Mismatches	3 (fixed)
Available	Online
Year	2013

ZiFiT was originally published in 2013 as a tool for enabling identification of potential *zinc finger nulcease* (ZFN) sites in specific target sequences. Later extended by support for the *CRISPR/Cas* taget sites identification and construction of *gRNA*. The application is available online through web interface.

ZiFiT supports only one to one design, one *gRNA* to one *target site* (this is common for all tools with few exceptions which will be mentioned). Also is able to design a pair of *gRNAs* for a double strand approach.

The application input is single or multiple *FASTA* sequences. *ZiFiT* only scan some regions of the genome for off-targets up to 3 mismatches. The maximum number of mismatches is fixed and cannot be adjusted. The application supports only one PAM sequence `NGG`.

Output of the application is a list of possible target sites with found *off-target* sites. Each off-taget site has a position where occurs, a number of occurrences and highlighted mismatch position.

A list of currently supported species: Human, mouse, zebrafish, rat, mosquito, fly, roundworm.

3.2.2 CRISPR Design

Author	Zhang Lab, MIT [RHW+13]
Web page	`http://crispr.mit.edu/`
Mismatches	5
Available	Online
Year	2013

An application *CRISPR Design* is mainly notable for being the first published tool for *sgRNA* design. The *CRISPR Design* tool is a web application created to simplify the process of *CRISPR* guide selection in an input DNA sequence.

Input of the application is a sequence (or sequences), the user e-mail address and a custom name of a search. A link to the results is send to the e-mail adress after the evaluation of the input. The output contains a list of *target sites* with score and corresponding *off-target* sites.

The *CRISPR Design* uses prior information about occurrences so it does not guarantee to find all *off-target*. The application supports only one PAM sequence `NGG`. *CRISPR Design* uses *off-target* scoring based on [HSW+13].

A list of currently supported species: Human, mouse, zebrafish, c. elegans, rat, fly, rabbit, pig, possum, chicken, a. thaliana, dog, mosquito and stickleback.

3.2.3 Cas9 Design

Author	Ming Ma, Adam Y. Ye, Weiguo Zheng, and Lei Kong [MYZK13]
Web page	`http://cas9.cbi.pku.edu.cn/`
Mismatches	1
Available	Online
Implementation	Java, Bowtie, Tabix (SAMTools), Vienna RNAfold
Year	2013

Tool introduced by [MYZK13]. Authors created a database of *gRNAs* and their corresponding efficiency with many elements freely available. Application is written in Java programing language and Tomcat web server.

The first step is to obtain all candidate sequences from the input based on the $N_{20}NGG$ template pattern, that represents all possible combinations of 20-mer sequence ending with `NGG` PAM sequence, by using Java regular expression matching. In the second step, program puts all candidate guides to a FASTA file and try to map them onto target genome using the *Bowtie*. The next steps are check for any *SNPs* and predicting RNA secondary structures for candidate guide RNA. The output is similar to previous tools, a list of possible *target sites* with location on the genome. The *Cas9 Design* does not look for the *off-targets*.

Cas9 Design tool provides limited features compared to the alternatives. For example the maximum number of mismatches is only 1.

A list of currently supported species: Human, mouse, zebrafish, rat, fruitfly, tomato, sheep, honeybee.

3.2.4 CasOT

Author	PKU Zebrafish Functional Genomics Group, Peking University [RHL+13, MAS+13, XCK+14]
Web page	`http://eendb.zfgenetics.org/casot`
Mismatches	6
Available	Offline (Windows, OSX, Unix)
Implementation	Perl script
Source code	`http://eendb.zfgenetics.org/casot/download.php`
Year	2014

CasOT is a Perl script that can run on any main platform (Windows, Unix systems or OS X). The application provides searching mode as single gRNA design, paired gRNAs (double strand approach) design and a target-and-off-target searching mode.

In the first two modes the application input is a single *target site*. The third searching mode is for helping to design *target sites* from a sequence up to 1kb.

The input for the first two methods is a FASTA sequence x of 21 up to 33-nt symbols, where last three symbols are a PAM sequence and a genome sequence t file to search for off-target sites. We can find several widely used genomes available at the official website. A user can also specify several parameters like PAM type and the maximum number of mismatches.

The program reads input sequence x, parses it into a hash table that contains all possible 12-nt long sequences similar to the x with up to user defined mismatches. The genome sequence is than scanned linearly to find any occurrence of sequences in the hash table. For any found occurrence of the off-target site, check if PAM sequence match and then calculate the off-target score, highlight mismatches positions and add to the output.

The *CasOT* supports any species, we just need a local copy of the genome sequence.

3.2.5 Cas-OFFinder & Cas-Desiner

Author	Bae, Sangsu and Park, Jeongbin and Kim, Jin-Soo [BPK14]
Web page	`http://www.rgenome.net/cas-offinder/`
Mismatches	9 (Online version only)
Available	Online and offline (Windows, OSX, Unix)
Implementation	C++, OpenCL
Source code	`https://github.com/snugel/cas-offinder`
Year	2014

The *Cas-OFFinder* was introduced by [BPK14]. This application is available online through web interface and also as offline command line version. The application is written in C++ and OpenCL and so requires OpenCL device to run properly.

The online version of the application supports up to 9 mismatches, requires to input query sequences with length between 15 to 25 nt in FASTA format. A PAM sequence we can choose from six types (`NGG`, `NRG`, `NNAGAAW`, `NNNNGMTT`, `NNGRRT`, `TTTN`). The online version supports almost 90 species for genome sequence.

The command line tool supports technicaly unlimited number of mismatches. The input of the application is a genome sequence and an input sequence. The *Cas-OFFinder* alternates between C++ and OpenCL. First, the application reads genome sequence and prepare the data for the second stage, that includes deviding the data into units of smaller size to overcome the memory limitation. Second, an OpenCL code searches through entire genome to find all PAM sequences. The next stage collects the informations about specific sites containing PAM sequences and delivers to the next stage, which counts the number of mismatched for each sites. The last stage selects potential off-targets based on number of mismatches.

The *Cas-Designer* is a Python script upon the *Cas-OFFinder*. The input is sequence where *Cas-Designer* find all possible guides and for each guide run the *Cas-OFFinder*.

One of the many features of the *Cas-OFFinder* is that it scans entire genome to find possible off-target sites, but disadvantage is lack of score of the *off-targets* neither of the *target sites*.

3.2.6 CHOPCHOP

Author	Tessa G. Montague, José M. Cruz, James A. Gagnon, George M. Church, and Eivind Valen [MCG⁺14]
Web page	`https://chopchop.rc.fas.harvard.edu`
Mismatches	2
Available	Online
Year	2014

The *CHOPCHOP* is a web based application with many advanced options. The *CHOPCHOP* accepts three forms of input: gene name, genomic coordinates or DNA sequence. A gene name is converted into genomic coordinates in the selected organism using various sources (for example [UCS]). Genomic coordinates are converted into the DNA sequence and if the user provides DNA sequence it is scanned for all potential *target sites*. The *off-targets* are searched using the *Bowtie* tool with up to 2 mismatches. Each sgRNA is scored based on three criteria. The number of off-targets, the number of mismatches within the off-targets and presence of `GC` at the start of off-targets.

The output is a list of possible *guides* within input sequence with highlighted mismatches, position and a sequence. *CHOPCHOP* operates only on the input sequence.

A list of currently supported species: Human, mouse, zebrafish, Drosophila melanogaster, c. elegans, Saccharomyces, Arabidopsis thaliana, Xenopus tropicalis, rat.

3.2.7 Benchling

Author	Benchling, Inc.
Web page	`https://benchling.com/`
Available	Online
Year	2014

The *Benchling* is a commercial tools with a lots of features not only for gRNA design. Designing features include single or paired gRNA design, off-target and on-target scores, but only from 16 to 24-nt long guides and limited option of the PAM sequences. Also number of mismatches can't be adjusted neither is known.

The *Benchling* is not only for CRISPR/Cas9 design but provides tools for other fields of biology as well. The users can also write all their research into build in notebook. All these features make *Benchling* one of more promising commercial options that provides a whole laboratory workbench.

The user interface is realy smooth and intuitive even for Informatic. 7 PAM sequences are supported and over 50 species. For the off-target score they use the same method as the *CRISPR Design* tool and also refers to it as a source.

3.2.8 GT-Scan

Author	O'Brien, Aidan and Bailey, Timothy L [OB14]
Web page	`http://gt-scan.braembl.org.au/gt-scan/`
Available	Online & Offline
Mismatches	3
Implementation	Python, Bowtie
Year	2014

GT-Scan is a web based application. The input sequence, template of guides, number of mismatches and off-target filter can be specified as the parameters of the application. The application scans an input genomic sequence for candidate targets and ranks them in terms of the number of exact or approximate off-targets in the genome the same way as the other applications.

The only supported PAM sequences are `NGG` and `NAG`. The *GT-Scan* first step is to convert the target rules into regular expression and then finds all the candidate targets in the input sequence using regular expression. For each possible *guides* generates a list of all possible *off-targets* that have up to 3 mismatches from the *guide* and apply the user defined off-target filter to reduce the number of possible off-targets. The *GT-Scan* combines all off-targets for all target guides into a single list of sequences and uses the Bowtie to identify matches in the genome sequence. The output contains all possible guides with scored off-targets.

The offline version is a local web server that can be installed on any computer and is opearted through web browser.

A list of currently supported species: Human, mouse, zebrafish, chimpanzee, mouse, cat, dog, horse, pig, guinea pig, pika, rabbit, rat, tasmanian devil, chicken, zebra finch and 16 others.

3.2.9 CCTop

Author	Stemmer, Manuel AND Thumberger, Thomas AND del Sol Keyer, Maria AND Wittbrodt, Joachim AND Mateo, Juan L. [STdSK$^+$15]
Web page	`http://crispr.cos.uni-heidelberg.de/index.html`
Available	Online
Mismatches	5
Implementation	Python, Bowtie
Year	2015

The *CCTop* is a web based application with a lot of parameters that the user can define. Besides the usual input as an input sequence, a PAM sequence and a genome sequence, the user can also specify a lot of other informations about the *guides* and the *off-targets*. The both sides of a guide can be limited by two symbols (nucleotides), lenght of the guide from 15 to 22 and *off-targets* filter.

The *CCTop* supports 22 species for the guide design.

We covered only a fragment of a CRISPR design tools. Sites like [Add, OMI] maintaine a list of existing tools.

3.3 Compare of Existing Tools

To compare the existing tools we used a total elapsed time of each application. We started to measure the time as the application started and stopped when the output was completed. This type of measurement was used because most of the applications are web based and therefore, another method was not applicable.

We must take into account that not all applications are using the same versions of the genomes or that some of them may have the genome loaded into the main memory already or provide different output. The results may not have the corresponding value. More accurate comparison would require a biologist who would divide applications into the groups of the same type and then compare them within one group.

We used the same input data, parameters and additional options for all testing applications. The input sequence was `GGAGCTGCAGGGACCTCCATGTCCTGGGACT GTTTGTGCAGGGCTCCGAGGGGACCCATGTGGCTCAGGGTGGCTAAGGGGGCAATGCTGCCCCC ACCCGCTGGATGAC`, the input guide was `CTGTTTGTGCAGGGCTCCGA`, the number of mismatches set to 3 where possible, the *PAM sequence* `NGG` and the *off-targets* were searched in the human genome.

ZiFiT We used the input sequence. Does not provide score of the possible guides, does not search entire genome for *off-targets*. Lenght of the *guide* is 20-nt. In the input sequence, *ZiFiT* founds in total 7 *guides* and 180 *off-target sites*.

CRISPR Design We used the input sequence. Lenght of the *guides* is 20-nt. The 22 guides were found and tested with an average of 300 *off-targets*.

Cas9 Design We used the input sequence, 20-nt guide lenght. The application only scan input sequence for possible guides sites and try to map them into the genome sequence without any scoring. 22 guides found.

CasOT We used the input sequence. The number of possible mismatches was set to 3, the *target-and-off-target* searching mode. Other parameters have default values. Building of the hash table for the input sequence took 1 second.

Cas-OFFinder & Cas-Designer The online versions: The input for *Cas-OFFinder* was single guide. The input for *Cas-Designer* was the input sequence.

The offline versions: The input for *Cas-OFFinder* CPU version was a single guide. The GPU version and *Cas-Designer* coud not be tested.

For all versions the output contained only few *off-targets*.

CHOPCHOP We used the input sequence. All parameters were left with default values. 22 guides were found without a list of the *off-targets*.

Benchling We used the input sequence. The user can only specify some of the parameters: guide length, genome, PAM sequence and design type (single or paired). The output includes 22 guides with the on-target and off-target scores.

Application	Time
ZiFiT	4s
CRISPR Design	2m 25s
Cas9 Design	6s
CasOT	17m 35s
Cas-OFFinder (online)	9s
Cas-Designer (online)	7m 5s
Cas-OFFinder (offline CPU)	5m 26s
CHOPCHOP	9s
Benchling	10s
GT-Scan	13s
CCTop	35s

Table 3.1: Time comparison of the CRISPR/Cas9 design tools.

GT-Scan We used the input sequence. The *GT-Scan* found 22 guides in the input sequence with an average of 35 *off-targets*. The off-targets have a score and highlighted position of mismatches.

CCTop We used the input sequence. No filter of the off-targets was used. The ouput contained 22 possible guides with (probably) the top 20 off-targets for each guide. Neither the off-targets nor guides were scored.

We have got a really not comparable results. The main time difference is in the search for the *off-targets*. The applications that do not score the off-targets neither do guides and not search entire genome for the off-targets have better running time, but may not be always accurate. Our goal is to find all possible off-target sites like *CRISPR Design*, *CasOT* and *GT-Scan*.

Chapter 4

Analysis and design

4.1 Approximate String Matching

To solve the given problem of scoring guides and their off-targets we need to be able to search for the guide in the text with any number of mismatches. The one method is to build an index over the input text that directly supports search query with mismatches. This is usefull mainly for short texts and small number of mismatches. Possible choice for such index is to use a finite automaton. The finite automaton is a computing model that can be represented as a graph. However, the finite automaton is constructed for the specific number of mismatches and the size of resulting structure can be huge.

A finite automaton is an computational model. Its purpose is to determine whether an input word belongs to a *language* described by the finite automaton. The automaton for the text t and m mismatches can accept all prefixes of the text t with possible m mismatches. We also want to search for any substring of the text and at this point the automaton solution is simply not practical.

The better solution for our problem is to use an index build over the text t without mismatches, then generate all possible strings up to m mismatches from the pattern and search each of them using the exact index.

4.1.1 m-neighborhood

A m-neighborhood of a string w is defined as a set of all strings with m and fewer mismatches from the w. We will denote the m-neighborhood set of the string w as $N_m(w)$.

The advantage of this solution is that we can use indexes for exact string matching, like *FM index* with succinct representations of data structures, that have smaller size than the original text. M-neighbourhood $N_m(w)$ is generated and all strings of $N_m(w)$ are searched for using the exact index.

The number of mismatches between two string w and v is a *distance*. The *SNP* in biology is equal to the definition of a *Hamming distance*. The

$$N_m^H(w) = \sum_{n=0}^{m} \left(\left(\frac{|w|!}{n!(|w|-n)!} \right) (|\Sigma| - 1)^n \right) \tag{4.1}$$

Figure 4.1: The number of strings with Hamming distance up to m from a string w

m	Size of $N_m^H(w)$
0	1
1	81
2	3121
3	76081
4	1316401
5	17192497
6	175953457
7	1446041137
8	9701611057
9	53731317297
10	247462024753
	$\vdots$

Table 4.1: The size of the $N_m^H(w)$, $|w| = 20$

Hamming distance is defined for two strings of equal length and it is the number of positions at which corresponding symbols mismatch. The m-neighborhood for the string w using the Hamming distance will be denoted as $N_m^H(w)$. The $N_m^H(w)$ is a set of all possible *off-targets* for a guide w.

The maximum number of elements from a $N_m^H(w)$ is equal to Equation 4.1. In Table 4.1 it is shown how many possible strings can be in the $N_m^H(w)$, where $|w| = 20$. Figure 4.2 shows the size of a 5-neighborhood ($N_5^H(w)$) of a string with a length l.

As we can see from Table 4.1, the size of a $N_m^H(w)$, even for relatively short strings, is growing rapidly and so the number of the off-targets we need to process. The time complexity of the *FM index* count query is $\mathcal{O}(|p|)$, for pattern p, which is for short pattern really good. However, if we have guide w, $|w| = 20$, and number of mismatches $m = 5$, we need to generate and score over 17 million possible off-targets of the guide w.

4.2 De Bruijn Graph

The *De Bruijn graph* (*DBG*) over the text t contains all k-mers that occur in the t. If an edge e from node i to node j exists in the *DBG*, we know that in the text must be a substring of the length $k + 1$ (($k + 1$)-mer) that equals to

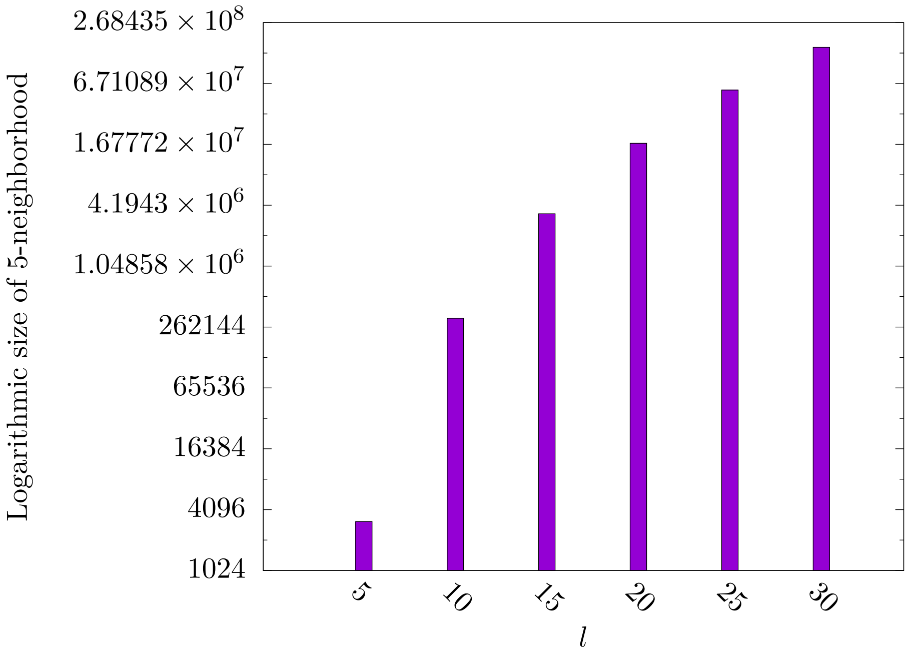


Figure 4.2: The size of the $H_5(w)$, $|w| = l$

the label of node i with appended symbol e to the end.

Let E_i be a set of edge symbols of a node i and L_i be a label of the node i. The E_i contains all symbols that occurs in the text after the L_i. Let G be the DBG over the text t of k-mers. The number of all $(k+1)$-mers of the t is equal to the $\sum_{i=0}^{N-1} |E_i|$, where N is the number of the nodes in the G. Therefore, if we follow all edges from all nodes we will generate a set of all $(k+1)$-mers and we can be sure that it is exactly the same $(k+1)$-mers of the text. However, if we want $(k+j)$-mers, $j > 1$, we have to traverse all nodes the G j times and at this point we can't guarantee that generated set of $(k+j)$-mers will contain only the substrings of the t.

The number of the nodes of the DBG increase depends on the choosen k as shown in Figure 4.3 and Figure 4.4 shows how the number of edges increase. The input files are created from the real DNA sequence of the human genome.

The maximum number of nodes of the DBG of k-mers is equal to Equation 4.1 where $|w| = k$, or simply to $|\Sigma|^k$. However, as we can see in Tables A.1 and A.2, for the DNA data the actual number of nodes is smaller already for $k > 3$. For example all 9-mers can theoretically have 1953125 nodes, but for all input files we observed the number about 265000, which is nearly 7.5 times smaller. And even the number of edges is lower than the theoretical maximum.

In Figure 4.5 we can see ratio between the number of edges to the number of nodes. This shows that DBG can be a very efficient filter for DNA sequences.

Besides the ratio, another interesting observation from the measured data

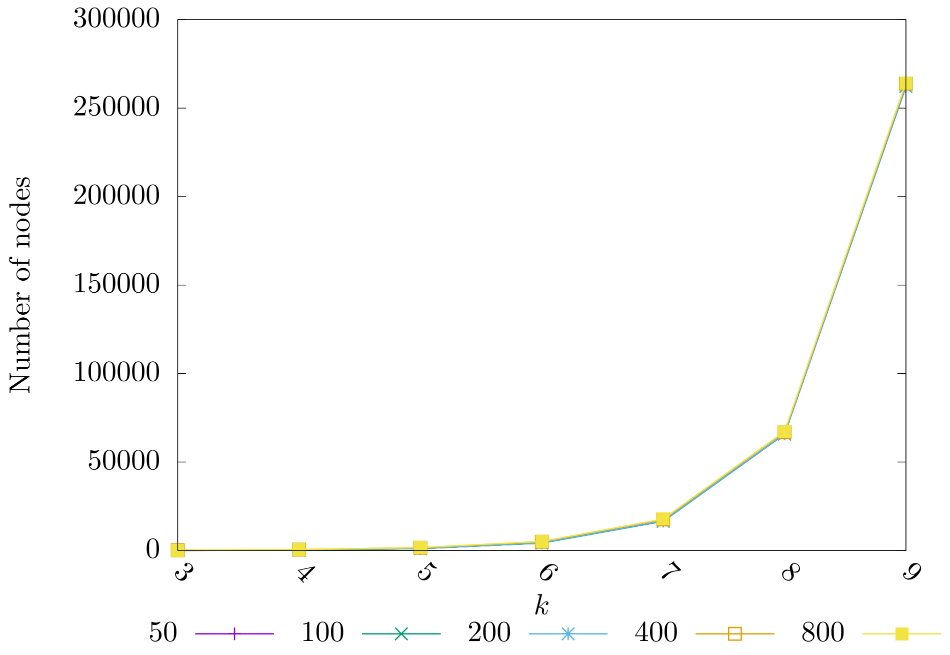


Figure 4.3: Number of nodes of the *DBG* for various k and input file size. Each line represent one input file. Name of the input file is *dna.**X**MB* where X is label of the line.

is that for all input files the number of nodes and edges is still about the same. And as we previously mentioned the number of nodes of the $(k + 1)$-mers is equal to the number of edges of k-mers (and vice versa).

All measured data used for visualizations are attached in appendix Section A.1. The number of the all k-mers in the genome can be approximated as shown in [CHG+09].

4.3 Design Proposal

Our motivation is to create an application of the index that supports the *count* query as fast as possible with the second criteria of space required to store and use the application. We also want to be able to filter some of the off-targets which certainly can not be in the text.

To create the fast index we simply pick one of the presented implementation of the *FM index* as it is small in size but still supports fast count queries. For the final aplication we use the *FM index*. However, all parts of the application can be replaced by another implementation with the same interface.

The information about what substrings are not in the text, would allow us to reduce the number of possible off-targets to be checked. We do not have that information, but we know what every k-mer of the text and every symbol

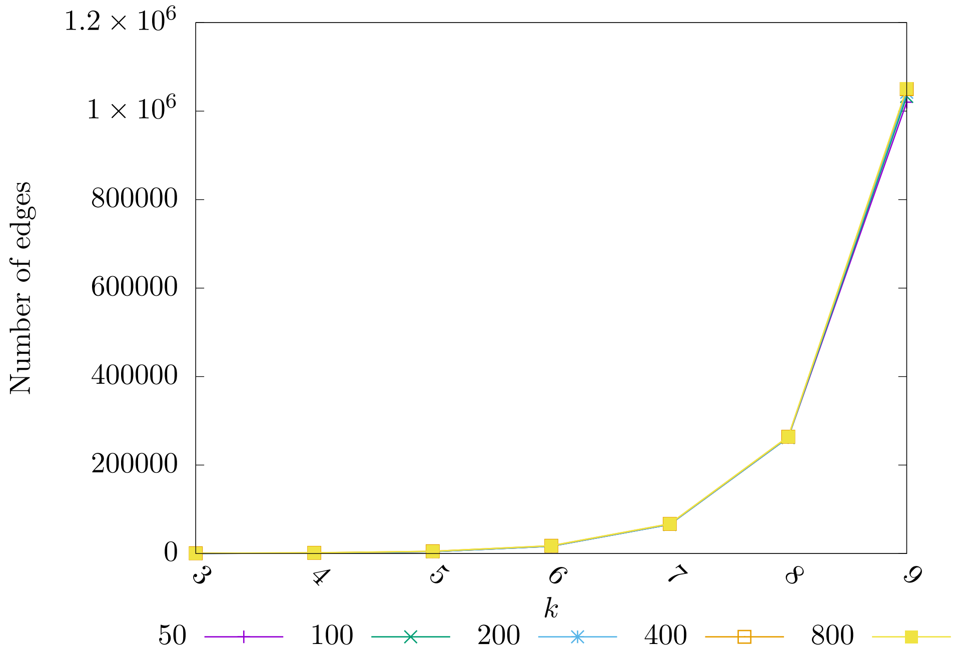


Figure 4.4: Number of edges of the *DBG* for various k and input file size. Each line represent one input file. Name of the input file is *dna.**X**MB* where X is label of the line.

that follows for all of the k-mers. So by traversing *DBG* we can generate sequence that is composed only of $(k+1)$-mers from the text.

The proposed solution for generating the off-targets, is by traversing the *DBG* of k-mers to generate strings of length n, $n \geq k$. This is why we want to have the smallest possible ratio between edges and nodes, because we would have less edges for node to follow when generating the off-targets.

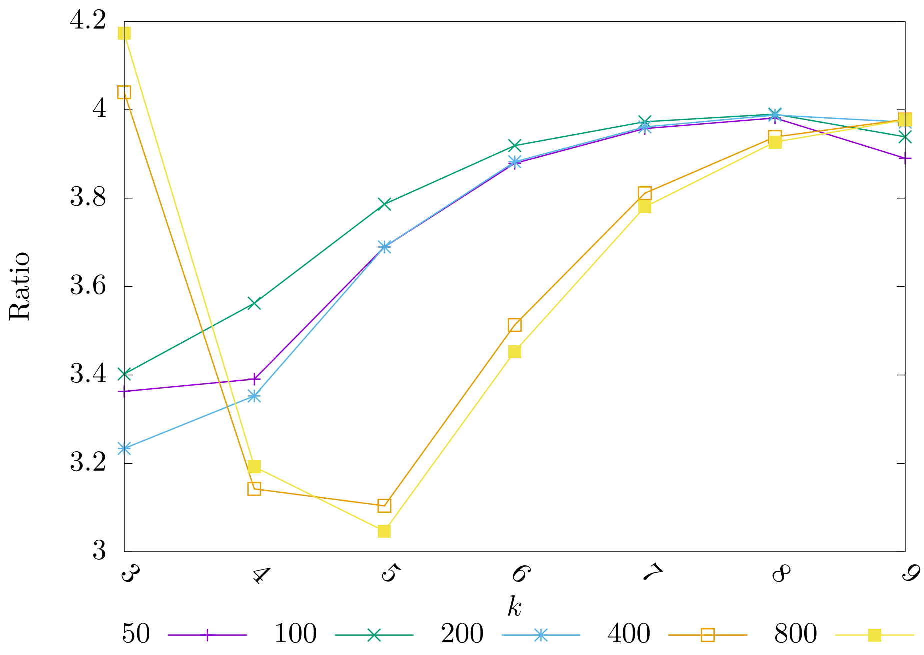


Figure 4.5: Ratio of the number of edges to the number of nodes. Each line represent one input file. Name of the input file is *dna.**X**MB* where X is label of the line.

Chapter 5

Implementation

5.1 Overall Structure

The core of the application is composed of two parts. The one that generates the k-neighborhood and the second part that finds the number of occurrences of the off-target using the *FM index*. The scheme of the application is shown in Figure 5.1.

The input of the application is one guide w, the number of possible mismatches m and the length of a k-mer to used by the *DBG* denoted as k.

The library *SDSL* [GBMP14] (*Succinct Data Structure Library 2.0*) developed by Simon Gog was used to create the *FM index* over the input file. The *SDSL* contains various implementations of data structures and their succinct representations.

The performance and size of the *FM index* the most important part is used representation of the *WT* and the *SA*. For the *SA* creation, *SDSL* support two sorting algorithm. The first is based on [Mora] and operates only in the main memory. The second one was introduced by Simon Gog [BZGO13] as one of co-authors and its advantage is it uses external memory to be more main memory efficient, but also maintain relatively good construction time.

For generation of the off-targets we chose to compare the *simple* and *DBG* generation method. For the input guide w, $|w| = n$, we need to traverse the *DBG* $n - k$ times starting at every “opening” node and by following all edges from each of visited node. The opening node is a node i with label L_i that is equal to the prefix *pref*(k) of the guide w and its m-neighborhood.

For the input we currently support only alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{N}, \texttt{T}\}$ for the input texts and guides. The genomes in the FASTA format are commonly contains only this five symbols so it is sufficient. However, this restriction is easy to remove. Also the size of the input guides must be greater or equal to the length of a k-mer of the *DBG*.

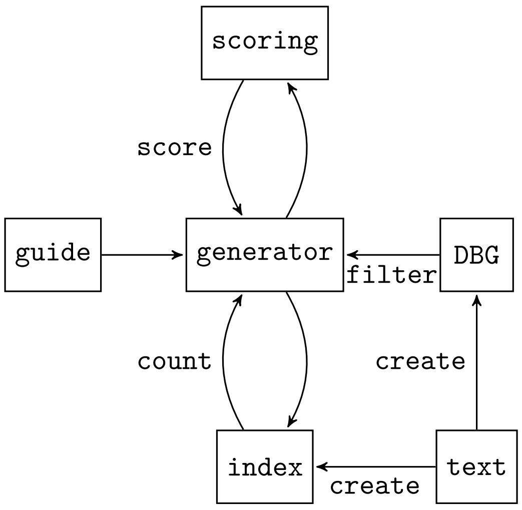


Figure 5.1: Structure of the application.

For the implementation we used the *C++* programming language. The *SDSL* library is also written in the *C++*. The application can be compiled and run on any major architectures for which the *C++* compiler can compile to.

5.2 Settings and Input

The application has only a command line interface. The configuration of the application is done via a configure file in the specific format. The format is shown in Example 5.1. The first line specifies the input guide w, the second line is the number of possible mismatches m and the last is the length of the k-mers of the *DBG* k.

Example 5.1 (Structure of the input file.)

```
CTGTTTGTGCAGGGCTCCGA
5
4
```

The settings of the application are done by three parameters. The parameter `-b` needs an input file name and it is indicating that the index and *DBG* will be build for the input file. The parameter `-i` needs an input file name, but it assumes that the *index* and *DBG* is already created. The last parameter `-c` specify the path to the configure file.

The calling Example 5.2 shows use of the parameters.

Example 5.2 (Use of the application parameters)

```
./application -b input -c config_file
./application -i input -c config_file
```

5.3 Index

The index is build over the *WT* with the *rank & select* support. The shape of the *WT* is based on the *Huffman coding*. The *Huffman coding* uses the number of occurrences to assign a code to the symbol. If the symbol occurs more often, Huffman coding will assign a shorter code to it. In the *WT* it is equal to the "path" from the root to the leaf of the symbol. However, our alphabet contains only 5 symbols so the shape is more likely to be balanced anyway, but the Huffman shape performs best. For the alphabet with only 4 symbols the *WT* has only two levels of the nodes, but with 5 symbols we need one more level for two least occurrence symbols. The size of the alphabet affects the overall speed of the application and different shapes of the *WT* can performs the time.

The *rank* support is build on top of simple bit vector with the superblock size of 2048 bits and each superblock is subdivided into the 5 blocks.

For the count query we need only the constructed *WT* from the *SA*. So to save the memory, we do not store the original *SA* as part of the index.

5.4 Off-target Generation

To be able to score the input guide, we need to know the number of occurrences of its all possible off-targets. The set of the off-targets is equal to $N_m^H(w)$.

The input guides can be of any length, but the most common length of the guides is 20-nt and mismatches up to 5. We know that for the string w, $|w| = 20$, the size of the $N_m^H(w)$ is equal to more than 17 millons of the possible off-targets. We naturaly feel that this number is too large, so out goal is to reduce the size while maintain the same running time. Another problem is to actualy generate and even operate with the entire set $N_m^H(w)$.

5.4.1 Using Recursion

A straightforward solution is to generate all possible off-targets, store them into the set then iterate through the set and score each off-target and as the last step to score the input guide based on the score of the off-targets.

For generation of the off-targets, we can use a recursive algorithm. Starting generation from the position 1 of the guide w to the last position and generated string is denoted as o. Possible symbol we can use at every position is any from the alphabet Σ. We choose one of the symbols for the position i and recursively call the same function to process $i + 1$ position. After we generate string of the length $|w|$ we add it to the set of the off-targets. The mismatch is at the position i where $o_i \neq w_i$. We can track the number of mismatches while generation of the string o and if we reach the maximum number of mismatched before the $|w|$ position, the string o is not from the $N_m^H(w)$.

The recursive algorithm is very similar to the *Depth-First search*, where we traverse a graph by starting at the root node and explores it as far as possible with the backtracking.

The general problem when generating the off-targets is the size of the set. If we want to generate the $N_5^H(w)$, $|w| = 20$, the size of the set is $17192497 * string\_size$, where the $string\_size$ is equal to 20 B. The resulting size is approximately 50 MB. For 6 mismatches and the same input guide, the amount of memory needed is nearly 440 MB. However, instead of storing all of them, we can directly score any found off-target and store only the accumulated result.

We will refer to this method as *simple method*.

5.4.2 Using DBG

The solution we proposed use the *DBG* to reduce the generated off-targets. Let G be the *DBG* of a k-mers, the input guide w and the number of mismatches m.

The first step is to find all *opening* nodes for the input guide. To do it, we generate the $N_m^H(\mathit{prefix}(k))$, the m-neighborhood of the prefix of the guide of the length k. For all k-mers from the $N_m^H(\mathit{prefix}(k))$ we try to find if the k-mer is in the *DBG*. If yes, we store the k-mer with the information about how many mismatches it already contains to the set T. All k-mers from the T are opening nodes.

The generation of the off-targets continue by traversing the *DBG* from the opening node in the T as a starting node s. We follow all edges from the node s to get to the $|E_s|$ new nodes. Every node expansions defines position i of the off-target and i is number of expansions from the s to the current node. The edge symbols in E_{s+i} are candidates for the position i in the off-target

The generation use the same technique as the *simple* generator for the number of mismatch and lenght of the off-targets.

5.5 DBG Construction

To construct the *DBG* we need to scan the input text for every k-mer. We use a sliding window of size k and linear pass through the text. For every sliding windows position we take current k-mer and check if is already in the *DBG* and if not we add. The last step is to add the edge symbol that is right after the current k-mer.

After the *DBG* construction, we need to interate through the nodes and mark edges of the nodes that if we follow we get into the same node. From the nodes and edges we then create the succinct representations, where the nodes are represented only as the bit array *last* and the table F of the symbols occurrences. The edge array is stored as the *WT*.

The time complexity of the *DBG* construction depends on the size of the k-mers set and the input text length. Because we need to have only one node for every k-mer, we need to for every position of the sliding window check if the k-mer is already in the set.

Time complexity of sliding through the genome is $\mathcal{O}(n \cdot k)$, because of the shift of the sliding window at every n position of the input text. Managing the set of nodes and edges takes $\mathcal{O}(\log \nu)$ in every step, where ν is the number of all k-mers. Overall time complexity is $\mathcal{O}(nk \cdot \log \nu)$.

The *Cosmo* [Bow] library version *0.5.1* contain the implementation of the *DBG* that we use, but the construction from the text and index operation were not implemented. We also had to reimplemented some of the other functions to correctly support our representation.

Chapter 6

Experiments

6.1 About Experiments

The motivation of our work was to create the application that support DNA data. This brings us benefits in terms of small alphabet and repetitive sequences. We have redured the experiments to only DNA sequences. The common length of guides is 20 symbols long sequence of nucleotides and the number of possible mismatches is 5. We will use this combination as the input.

6.1.1 Environment

All tests and measurements were made on a 64bit computer running the *OS X* version *10.11.4* with 8 GB of main memory and up to 60 GB of external memory used as a *swap*. In Table 6.1 all informations are summarize.

OS	*OS X 10.11.4*
CPU	*Intel Core i5, 2.7 GHz*
Main memory	8 *GB*
Swap	60 *GB on SSD*

Table 6.1: Summary of informations about testing environment.

6.1.2 Dataset

As the input texts we used real sequenced human genome from which files with various size were created. Table 6.2 shows from which parts of the human genome the input files were created and their length n. Each of the input file is created from another part of the genome for more generality. We will refer to the input files by their size in a graph measurements.

All the input texts have an alphabet $\Sigma = \{A, C, G, N, T\}$. Also all testing guides are over the same alphabet.

Input file	Chromosomes	n
dna.50MB	22	50818468
dna.100MB	15	107043718
dna.200MB	3	198295559
dna.400MB	1, 6	404997317
dna.800MB	2, 4, 5, 21, X	806536819

Table 6.2: Input test files and corresponding chromosomes they were created of.

The human genome can be obtained from many sites, but we used data from the Ensembl [Ens]. The Ensembl is a public database for the genomes of many species. The guide sequence was taken from the random position within the genome.

6.1.3 Time measurement

To accurate measure the time we need to perform time measurement for the same input multiple times and take the average value to reduce the possible error.

The time itself was measured by *C* system function *getrusage* and wrapped see code 6.1. The function *getrusage* returns, among others, *user* and *system* time. The *user* time indicates how long the code itself run and the *system* time indicates how long code spends in the kernel (file operations, kernel calls, ...).

The sum of both user and system time gives us real time from the beggining of the measurement to its end.

Listing 6.1: Time measure function

```c
double getTime(void) {
    double usertime, systime;
    struct rusage usage;

    getrusage(RUSAGE_SELF, &usage);

    usertime = (double) usage.ru_utime.tv_sec +
            (double) usage.ru_utime.tv_usec / 1000000.0;

    systime = (double) usage.ru_stime.tv_sec +
            (double) usage.ru_stime.tv_usec / 1000000.0;

    return (usertime + systime);
}
```

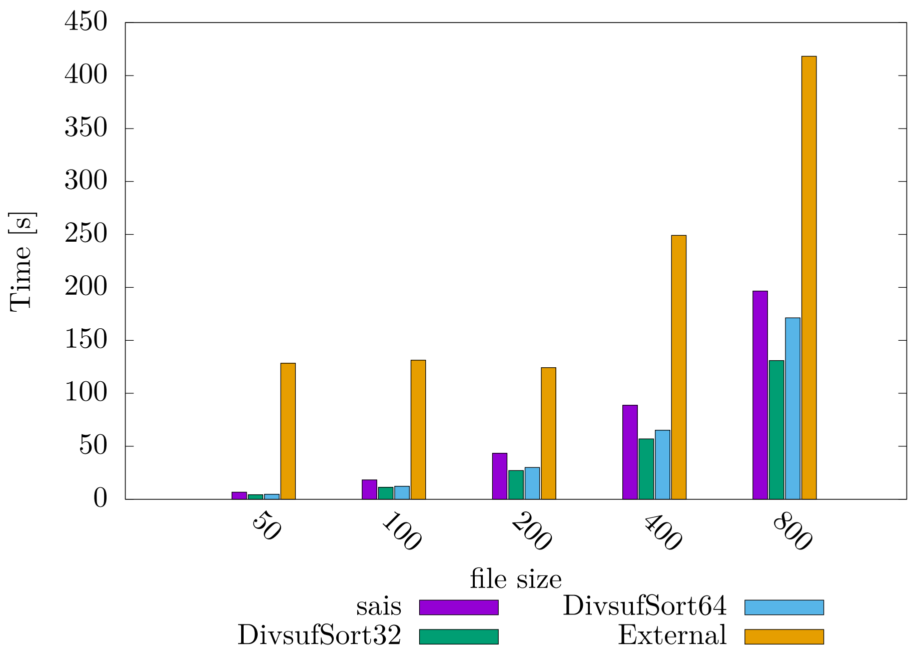


Figure 6.1: Time of suffix array construction using various algorithms.

6.2 Suffix Array

The construction of the *SA* is one of the most consuming part of the index creation. We compared four *SA* construction algorithms.

The first *SAIS* [Morb] by Yuta Mori that runs in $\mathcal{O}(n)$ worst-case time and uses maximum $2n$ or $4|\Sigma|$ of additional working space, where n is an input text. This algorithm was one of the first that achieved linear time construction.

The second *DivSufSort* [Mora] also by Yuta Mori that runs in $\mathcal{O}(n \log n)$ worst-case time and using only $5n + \mathcal{O}(1)$ additional space. The *DivSufSort* is currently one of the best known construct implementations for the *SA*. It uses divide and conquer principle and bucket sorting technique [1]. We have tested both 32 and 64 bit implementation.

The third construct algorithm [BZGO13] by Timo Beller and co-authors aims to be space-efficient that uses on average about $1.5n$ main memory and $10n$ external memory (hard drive space not swap).

Figure 6.1 shows time comparison of the construct algorithms and Figure 6.2 shows the difference between sizes of the input text and resulted SA.

We can see difference in hidden constant. The *SAIS* is slower than the *DivSufSort* even that the worst-case running time is asymptoticaly better.

[1]Brief introduction to the bucket sort can be found at Wikipedia: `https://en.wikipedia.org/w/index.php?title=Bucket_sort&oldid=707560846`

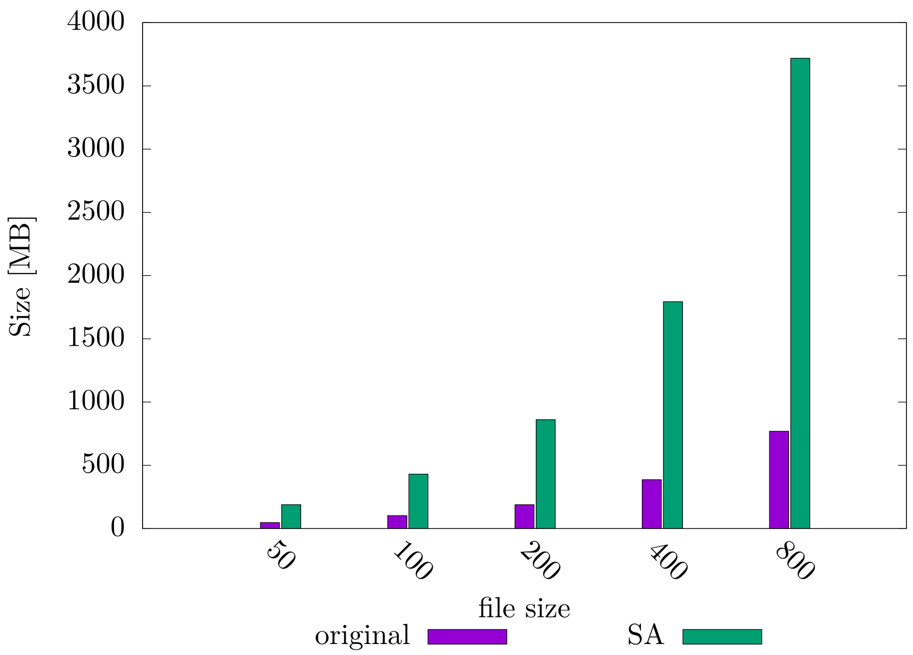


Figure 6.2: Size of the input text and the resulted SA.

6.3 FM Index and BWT

The actual time required to perform queries on the *FM index* greatly depends on used representations of data structures. If we want a small index size and fast query operations, the succinct representation is our best choice.

The input for all measurements of the performance of the *FM index* was used testing file of DNA from the *Pizza & Chili corpus* [FN]

6.3.1 Count

The count query mainly depends on the chosen representation of the *BWT* and *rank & select* data structures. We discussed use of the *wavelet tree* with *RRR* data structure previously and Figure 6.3 shows how the time of the count depends on chosen block size b of the *RRR* structure. We remind that block size directly affects how many of precomputed elements will be stored for the *rank & select* support. The *bitvector* in the figure is a simple bit vector representation that for the *rank* support builds similar blocks/superblocks scheme, but as two individual tables.

6.3.2 Locate

The locate query is more interesting because we need to use the *SA*. Instead of entire *SA* we used the *CSA* representation. Table 6.3 shows how the time of

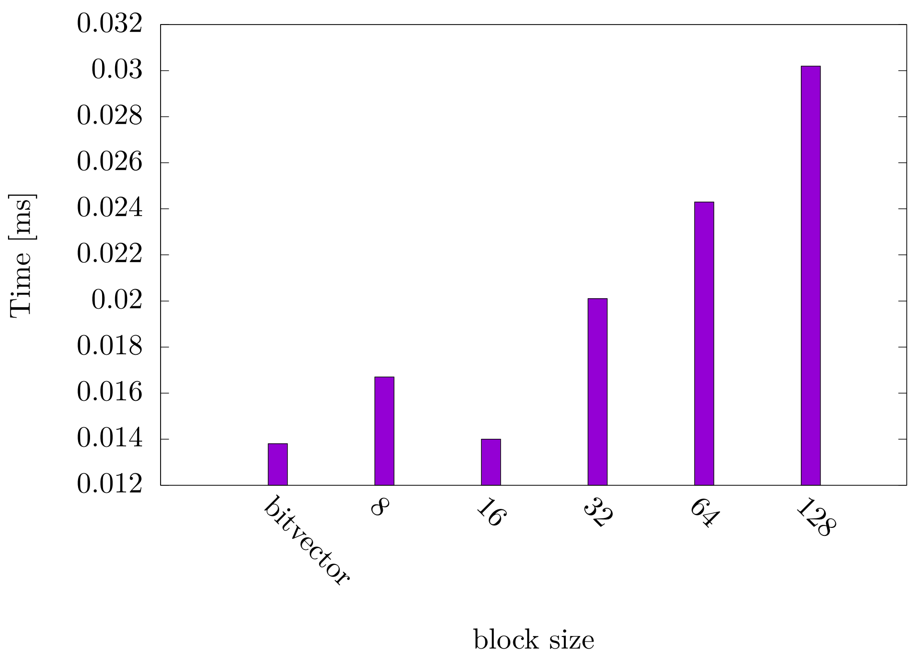


Figure 6.3: Time of the count query for *RRR* with the block size b.

the query varies depending on the sample rate of the *CSA* and also the size of created index.

sample rate	time [ms]	size [MB]
4	0.9	435
8	1.1	249
16	2.7	155
32	6.2	107
64	13.4	83
128	24.0	70
256	45.2	64
4	1.1	432
8	1.8	246
16	4.4	152
32	9.8	104
64	20.9	80
128	42.0	67
256	85.1	61

Table 6.3: Time of locate query and the size of the index.
The first half shows *WT* with each level represented using the *bitvector* and the second half using the *RRR* with block size $b = 16$.

6.4 De Bruijn graph

We have introduced algorithm to build the *DBG* by linear passage through the text and storing all unique k-mers with corresponding edges. The bottleneck of our current solution is maintain of the set of k-mers, because we do not want duplicity.

In Figure 6.4 we can observe the construction time of the *DBG* for k from 3 to 9. The lower k is meaning less and bigger k takes too much time. Figure 6.5 show the size of constructed succinct representation of the *DBG*.

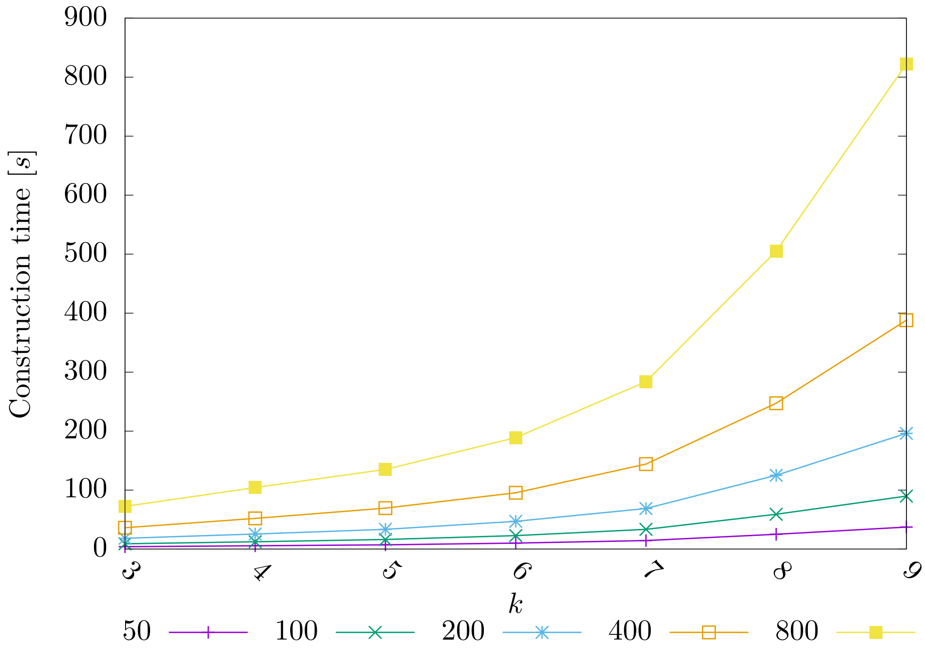


Figure 6.4: Time of the construction of the succinct *DBG* for various k and input file size.

6.5 Generation of the Off-targets

6.5.1 Simple Method

We have discussed that scoring off-targets is done within their generation to reduce the main memory requirements. The *simple* method will always generate the same set of the off-targets. So the time of the *simple* generator mainly depends on the number of mismatches and secondly on the index.

Table 6.6 shows how the time of generation and scoring depends on the number of possible m mismatches using the *DBG* with $k = 5$ (all 5-mers of

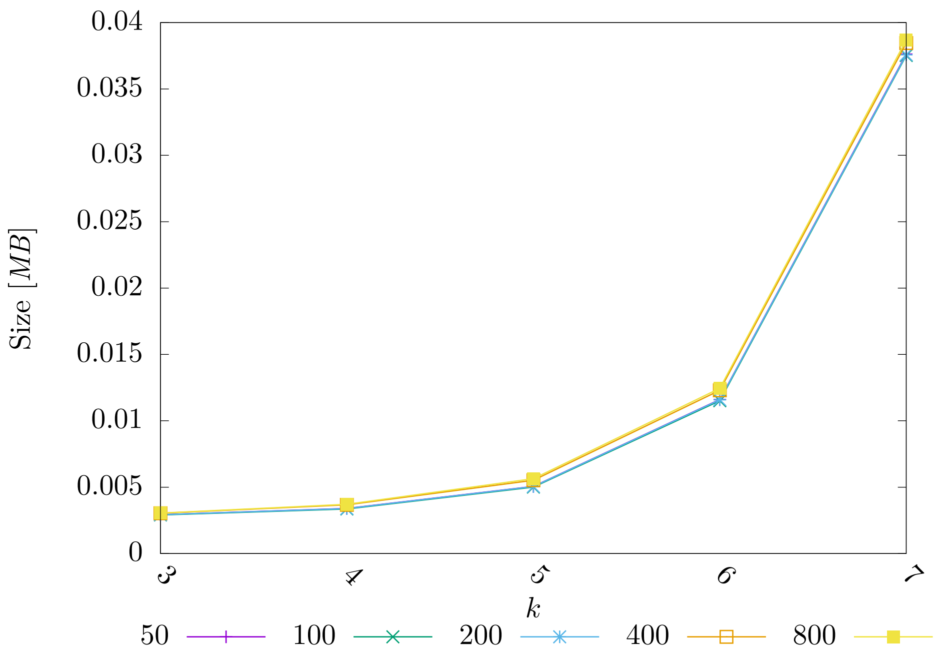


Figure 6.5: Size of the succinct *DBG* for various k and input file sizes.

the text). The time does not change much for other values of k and remains similar.

The number of generated off-targets by the *simple* method does not change at all. This is logical because it does not use any information about the input and simply generates entire m-neighborhood and can be see in Table 6.7.

6.5.2 DBG Method

The generation using the *DBG* of k-mers uses informations about the text to reduce the number of generated off-targets.

The time of the *DBG* methods depends on heavily on the input text and chosen k. To reduce the number of possible off-targets we need to choose k for which the *DBG* has the smallest possible *ratio* between edges and nodes (all nodes have the least number of edges). We then do not need to follow so many edges from each node and thus the set of the off-targets would be smaller.

The effect on the number of possible off-targets generated using the *DBG* method is shown in Figure 6.8 where $k = 5$ and in Figure 6.9 comparison between the *simple* and the *DBG* generator for the input dna.800MB and $m = 5$.

The time that *DBG* need to generate and score the off-targets is shown by Table 6.4 for various mismatches m. We can see that the time strongly depends on the number of possible mismatches and not so much on the input file size.

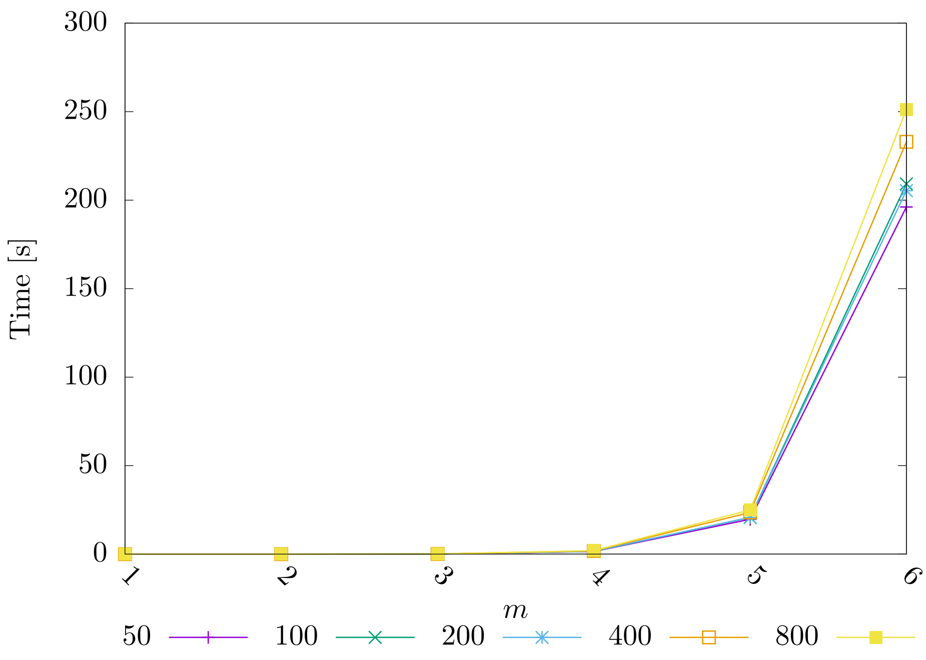


Figure 6.6: Time to generate and score off-targets using *simple* method.

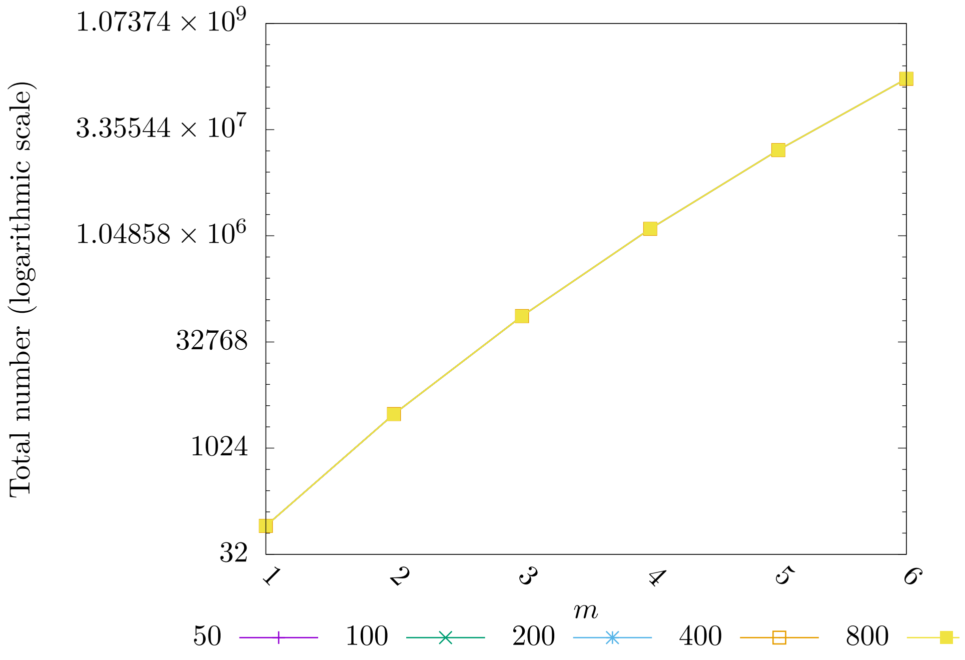


Figure 6.7: Number of generated off-targets by the *simple* method.

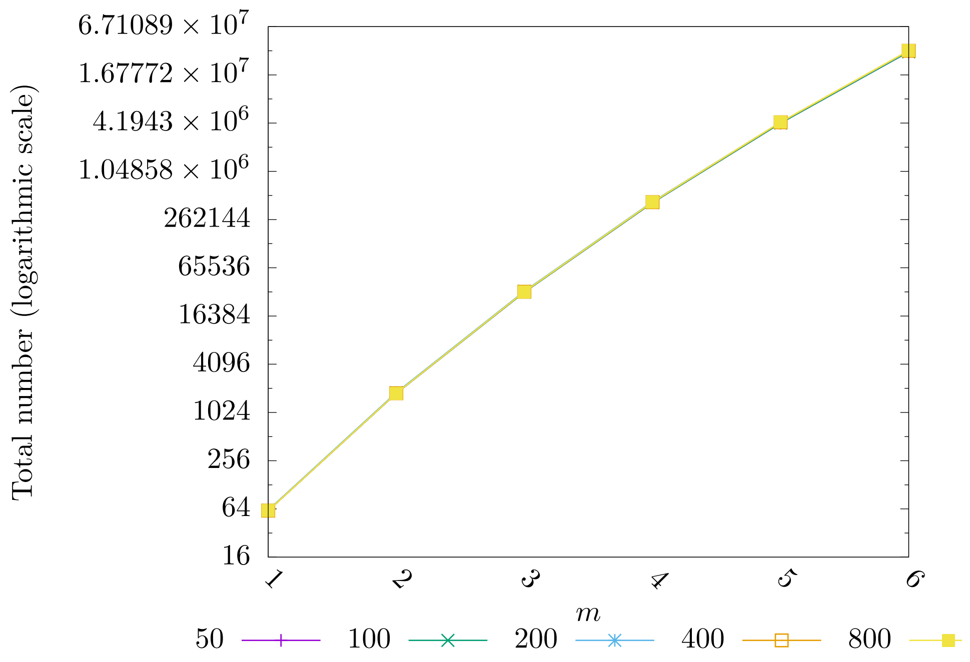


Figure 6.8: Number of generated off-targets by the *DBG* method and $k = 5$.

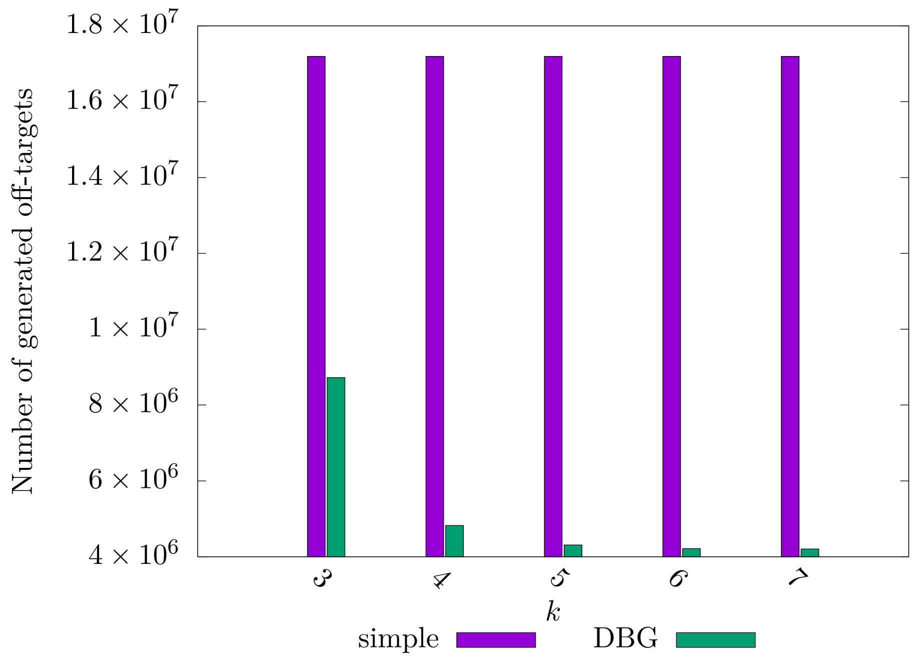


Figure 6.9: Number of generated off-targets by the *DBG* and *simple* methods and $m = 5$.

m	50	100	200	400	800
1	0.001251	0.002012	0.001441	0.001629	0.001791
2	0.024536	0.023509	0.022552	0.028701	0.040977
3	0.308728	0.311608	0.340084	0.354423	0.353836
4	3.32457	3.33958	3.60343	3.77214	4.07731
5	28.5578	29.0358	31.4737	32.0798	32.0626
6	188.924	197.577	205.613	219.085	231.679

Table 6.4: Total time of the *DBG* generator, $k = 5$

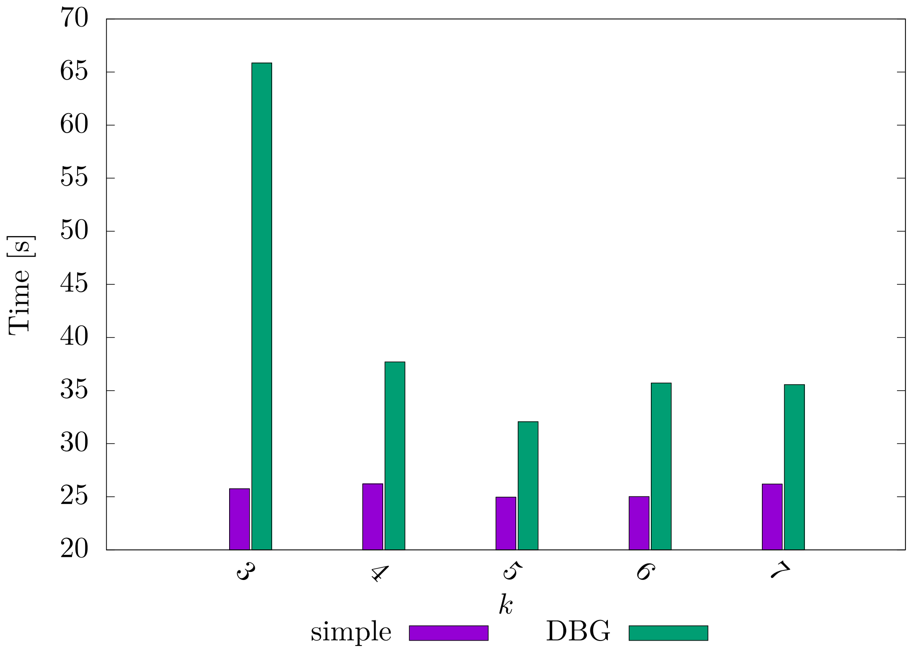


Figure 6.10: Time to generate and score off-targets using *simple* and the *DBG* method, $m = 5$.

This is due the fact that constructed *DBG* are very similar for all input files. The more interesting is comparison in Figure 6.10 that shows differences in time generators *simple* and *DBG* for the input file dna.800MB and $m = 5$.

The results are not satisfying, although the number of off-targets is reduced, the time of the *DBG* generator is above the *simple* generator which must score entire m-neighborhood.

We have tested the *DBG* generator for the k from 3 to 7 and possible mismatches m from 1 to 6. The best results were achieved by the *DBG* of 5-mers with the lowest generation time and generated number of the off-targets is comparable to the *DBG* with higher k.

The time result for the *DBG* method with $k = 3$ is also interesting, but not surprising. The input files dna.400MB and dna.800MB have the worst *ratio* for the *DBG* with 3-mers and this is the consequence of it. From the results we may say that the *ratio* above 4 is not practical for our problem.

The data shown in this section are for the best possible k or m. The test for other combinations are very similar to these presented, but they would cover a lot of pages with only graphs and tables. However, some of them are included in Section A.3.

Conclusion

My motivation to write the thesis in the bioinformatics field was the interest in combination of biology and computer science. The bioinformatics is a constantly evolving field where many problems are not efficiently solvable, mainly because lack of attention from the informatics. However, in recent years the situation has been changing and the mentioned field has become more attractive.

During writing the overview of the current tools used for *CRISPR* systems designing, I found out about the current state-of-art and that all applications are using very similar process. All the optimizations are done in the index to filter the regions which have a low probability that the guide or off-targets can occur in.

I focused on the possibilities of reduction of the number of the off-targets. The De Bruijn graph is great concept to know more informations about the input text that help to reduce the possible off-targets. Experiments show that the idea is promising and in the comparison to all the off-targets the number of generated off-targets using *DBG* was always smaller. Unfortunately, the generation time of the *DBG* method is currently higher than the *simple* method.

The application is successfully implemented to perform the scoring of the input guide for any input genome. All application modules are independent so that any can be replaced by another.

I successfully experimented with the *DBG* generator method for practical length of the k-mers and show optimal k and practical *ratio* of the edges and nodes. The current implementation works correctly for tested file sizes and may even work for larger files. However, the construction needs a lot more memory and time.

In the future work I suggest to focus on the *DBG* operations speed optimization at the expense of its size and construction time. Another interesting improvement would be to use the indexing method that dramaticaly reduces the size of the index for two or more genomes from the same species. This would create unique concept among *CRISPR* design tools which will support fast and small index of multiple genomes.

Bibliography

[Add] Addgene. The crispr software matchmaker: A new tool for choosing the best crispr software for your needs. http://blog.addgene.org/the-crispr-software-matchmaker-a-new-tool-for-choosing-the-best-crispr-software-for-your-needs. Accessed: 2016-04-23.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53 – 86, 2004. The 9th International Symposium on String Processing and Information Retrieval.

[BK03] Stefan Burkhardt and Juha Kärkkäinen. *Combinatorial Pattern Matching: 14th Annual Symposium, CPM 2003 Morelia, Michoacán, Mexico, June 25–27, 2003 Proceedings*, chapter Fast Lightweight Suffix Array Construction and Checking, pages 55–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[BOSS12] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de bruijn graphs. In *Proceedings of the 12th International Conference on Algorithms in Bioinformatics*, WABI'12, pages 225–235, Berlin, Heidelberg, 2012. Springer-Verlag.

[Bow] Alexander Bowe. Cosmo. https://github.com/cosmo-team/cosmo. Accessed: 2016-04-30.

[Bow11a] Alexander Bowe. Fm-indexes and backwards search. http://alexbowe.com/fm-index/, 2011. Accessed: 2016-01-03.

[Bow11b] Alexander Bowe. Rrr – a succinct rank/select index for bit vectors. http://alexbowe.com/rrr/, 2011. Accessed: 2016-04-16.

[BPK14] Sangsu Bae, Jeongbin Park, and Jin-Soo Kim. Cas-offinder: a fast and versatile algorithm that searches for potential off-target sites of cas9 rna-guided endonucleases. *Bioinformatics*, 30(10):1473–1475, 2014.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[BZGO13] Timo Beller, Maike Zwerger, Simon Gog, and Enno Ohlebusch. Space-efficient construction of the burrows-wheeler transform. In *Proceedings of the 20th International Symposium on String Processing and Information Retrieval - Volume 8214*, SPIRE 2013, pages 5–16, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[CB11] Thomas C. Conway and Andrew J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, 2011.

[CHG$^+$09] Benny Chor, David Horn, Nick Goldman, Yaron Levy, and Tim Massingham. Genomic dna k-mer spectra: models and modalities. *Genome Biology*, 10(10):1–10, 2009.

[CHL07] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings.* Cambridge University Press, New York, NY, USA, 2007.

[Cla98] David Richard Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Waterloo, Ont., Canada, Canada, 1998. UMI Order No. GAXNQ-21335.

[CN09] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE '08, pages 176–187, Berlin, Heidelberg, 2009. Springer-Verlag.

[CR13] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms for Molecular Biology*, 8(1):1–9, 2013.

[Ens] Ensembl. Ftp download - dna sequences. `http://www.ensembl.org/info/data/ftp/index.html`. Accessed: 2016-04-30.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium* *on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[FN] P. Ferragina and G. Navarro. Pizza and chili corpus - compressed indexes and their testbeds. `http://pizzachili.dcc.uchile.cl/`. Accessed: 2016-04-29.

[GBMP14] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[GGMN05] Rodrigo González, Szymon Grabowski, Veli Mäkinen, and Gonzalo Navarro. Practical implementation of rank and select queries. In *In Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05) (Greece*, pages 27–38, 2005.

[GGV03] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[GS12] Joseph Yossi Gil and David Allen Scott. A bijective string sorting transform. *CoRR*, abs/1201.3077, 2012.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pages 397–406, New York, NY, USA, 2000. ACM.

[HSW$^+$13] Patrick D Hsu, David A Scott, Joshua A Weinstein, F Ann Ran, Silvana Konermann, Vineeta Agarwala, Yinqing Li, Eli J Fine, Xuebing Wu, Ophir Shalem, Thomas J Cradick, Luciano A Marraffini, Gang Bao, and Feng Zhang. Dna targeting specificity of rna-guided cas9 nucleases. *Nat Biotech*, 31(9):827–832, 09 2013.

[Jac88] Guy Joseph Jacobson. *Succinct Static Data Structures.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1988. AAI8918056.

[KK14] Juha Kärkkäinen and Dominik Kempa. Lightweight external memory suffix array construction algorithm. In *In Proc. 2nd International Conference on Algorithms for Big Data*, ICABD 2014, pages 53–60, 2014.

[KKP15] Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. *Combinatorial Pattern Matching: 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 – July 1, 2015, Proceedings*, chapter Parallel External Memory Suffix Sorting, pages 329–342. Springer International Publishing, Cham, 2015.

[KS03] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

[LFB$^+$14] Bo Li, Nathanael Fillmore, Yongsheng Bai, Mike Collins, James A. Thomson, Ron Stewart, and Colin N. Dewey. Evaluation of de novo transcriptome assemblies from rna-seq data. *Genome Biology*, 15(12):1–21, 2014.

[LTC13] Felipe A. Louza, Guilherme P. Telles, and Cristina Dutra De Aguiar Ciferri. *Combinatorial Pattern Matching: 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, chapter External Memory Generalized Suffix and LCP Arrays Construction, pages 201–210. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[LTPS] B Langmead, C Trapnell, M Pop, and SL. Salzberg. Bowtie, an ultrafast memory-efficient short read aligner. `http://bowtie-bio.sourceforge.net`. Accessed: 2016-04-24.

[LTPS09] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3):1–10, 2009.

[MAS$^+$13] Prashant Mali, John Aach, P Benjamin Stranges, Kevin M Esvelt, Mark Moosburner, Sriram Kosuri, Luhan Yang, and George M Church. Cas9 transcriptional activators for target specificity screening and paired nickases for cooperative genome engineering. *Nat Biotech*, 31(9):833–838, 09 2013.

[MCG$^+$14] Tessa G. Montague, José M. Cruz, James A. Gagnon, George M. Church, and Eivind Valen. Chopchop: a crispr/cas9 and talen web tool for genome editing. *Nucleic Acids Research*, 42(W1):W401–W407, 2014.

[MM90] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[MM93] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[Mora] Yuta Mori. libdivsufsort. `https://github.com/y-256/libdivsufsort`. Accessed: 2016-04-12.

[Morb] Yuta Mori. Sa-is suffix array construction algorithm. `https://sites.google.com/site/yuta256/sais`. Accessed: 2016-03-22.

[Mun] Ian Munro. Succinct data structures. `https://cs.uwaterloo.ca/~imunro/cs840/SuccinctDS.pdf`. Accessed: 2016-04-14.

[Mun96] J. Ian Munro. *Foundations of Software Technology and Theoretical Computer Science: 16th Conference Hyderabad, India, December 18–20, 1996 Proceedings*, chapter Tables, pages 37–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[MYZK13] Ming Ma, Adam Y Ye, Weiguo Zheng, and Lei Kong. A guide rna sequence design platform for the crispr/cas9 system for model organism genomes. *BioMed Research International*, 2013(270805):4, 2013.

[OB14] Aidan O'Brien and Timothy L Bailey. Gt-scan: identifying unique genomic targets. *Bioinformatics*, 30(18):2673–2675, 09 2014.

[OMI] OMICtools. Crispr/cas9 tools. `http://omictools.com/crispr-cas9-category`. Accessed: 2016-04-23.

[OS06] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. *CoRR*, abs/cs/0610001, 2006.

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2), July 2007.

[RHL$^+$13] F Ann Ran, Patrick D Hsu, Chie-Yu Lin, Jonathan S Gootenberg, Silvana Konermann, Alexandro Trevino, David A Scott, Azusa Inoue, Shogo Matoba, Yi Zhang, and Feng Zhang. Double nicking by rna-guided crispr cas9 for enhanced genome editing specificity. *Cell*, 154(6):1380–1389, 09 2013.

[RHW$^+$13] F Ann Ran, Patrick D Hsu, Jason Wright, Vineeta Agarwala, David A Scott, and Feng Zhang. Genome engineering using the crispr-cas9 system. *Nat. Protocols*, 8(11):2281–2308, 11 2013.

[RRR02] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 233–242, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[sac] Suffix array construction benchmark. `http://homepage3.nifty.com/wpage/benchmark`. Accessed: 2016-03-22.

[SMR$^+$10] Jeffry D. Sander, Morgan L. Maeder, Deepak Reyon, Daniel F. Voytas, J. Keith Joung, and Drena Dobbs. Zifit (zinc finger targeter): an updated zinc finger engineering tool. *Nucleic Acids Research*, 38(suppl 2):W462–W468, 2010.

[STdSK$^+$15] Manuel Stemmer, Thomas Thumberger, Maria del Sol Keyer, Joachim Wittbrodt, and Juan L. Mateo. Cctop: An intuitive, flexible and reliable crispr/cas9 target prediction tool. *PLoS ONE*, 10(4):1–11, 04 2015.

[UCS] UCSC. Ucsc genome browser. `https://genome.ucsc.edu/`. Accessed: 2016-04-23.

[XCK$^+$14] An Xiao, Zhenchao Cheng, Lei Kong, Zuoyan Zhu, Shuo Lin, Ge Gao, and Bo Zhang. Casot: a genome-wide cas9/grna off-target searching tool. *Bioinformatics*, Jan 2014.

Appendix **A**

Data

A.1 De Bruijn Graph

k	50MB	100MB	200MB	400MB	800MB
3	102	92	107	127	127
4	343	313	346	513	530
5	1163	1115	1160	1612	1692
6	4291	4222	4280	5004	5155
7	16644	16546	16616	17579	17800
8	65870	65735	65816	66999	67287
9	262240	262285	262463	263872	264232

Table A.1: The number of nodes of *DBG*.

k	50MB	100MB	200MB	400MB	800MB
3	343	313	346	513	530
4	1163	1115	1160	1612	1692
5	4291	4222	4280	5004	5155
6	16644	16546	16616	17579	17800
7	65870	65735	65816	66999	67287
8	262240	262285	262463	263872	264232
9	1020172	1033087	1042516	1049665	1050900

Table A.2: The number of edges of *DBG*.

k	50MB	100MB	200MB	400MB	800MB
3	3.36275	3.40217	3.23364	4.0394	4.17323
4	3.39067	3.5623	3.3526	3.1423	3.19245
5	3.6896	3.78655	3.68966	3.1042	3.04669
6	3.87882	3.919	3.88224	3.513	3.45296
7	3.95758	3.97286	3.961	3.8113	3.78017
8	3.98118	3.99004	3.98783	3.9384	3.92694
9	3.89022	3.9388	3.97205	3.9779	3.97719

Table A.3: Ratio of the number of edges to the number of nodes.

k	50MB	100MB	200MB	400MB	800MB
3	0.0031271	0.00311184	0.0031271	0.00328732	0.00328732
4	0.00369167	0.00366879	0.00369167	0.00396633	0.00401974
5	0.0057745	0.00573635	0.00576687	0.00615597	0.00626278
6	0.0145006	0.014432	0.0144777	0.0149965	0.0151339
7	0.0574894	0.0573978	0.0574589	0.0580692	0.0582371
8	0.19711	0.187963	0.197332	0.198057	0.198256
9	0.661063	0.668688	0.674221	0.754849	0.755621

Table A.4: The size of succinct *DBG* [MB].

k	50MB	100MB	200MB	400MB	800MB
3	4.11351	9.04654	18.2553	36.1622	72.383
4	5.61691	12.3857	25.4802	52.1411	104.245
5	7.24875	16.3671	33.5676	69.3549	135.148
6	10.1439	22.8455	47.1628	95.5646	188.901
7	14.5103	33.5487	68.7645	144.011	283.758
8	25.1581	59.0816	125.034	247.287	505.028
9	37.3674	89.8241	196.153	388.072	822.302

Table A.5: Construction time of *DBG* [s].

A.2 FM Index

b	Time $[ms]$
bitvector	0.0138
8	0.0167
16	0.0140
32	0.0201
64	0.0243
128	0.0302

Table A.6: Time of count query for *RRR* with block size b.

A.3 Measurements

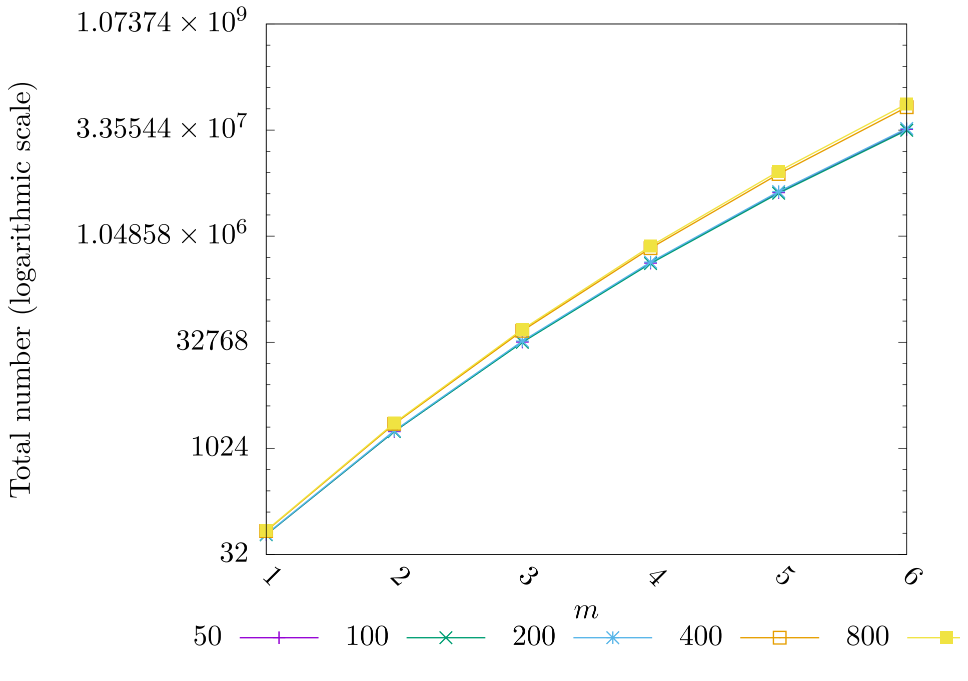


Figure A.1: Number of generated off-targets by the *DBG* method and $k = 3$.

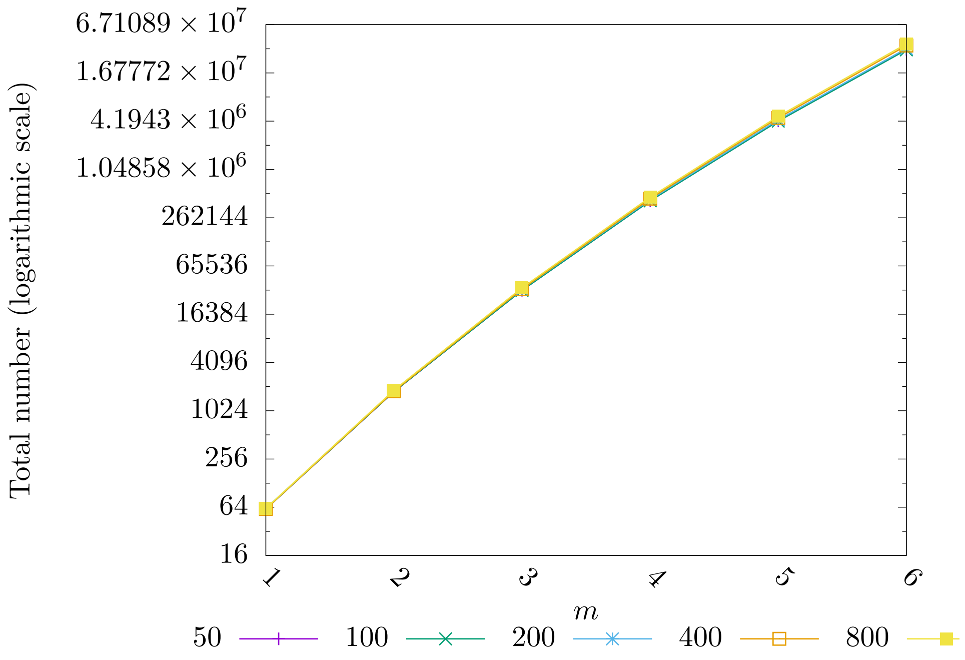


Figure A.2: Number of generated off-targets by the *DBG* method and $k = 4$.

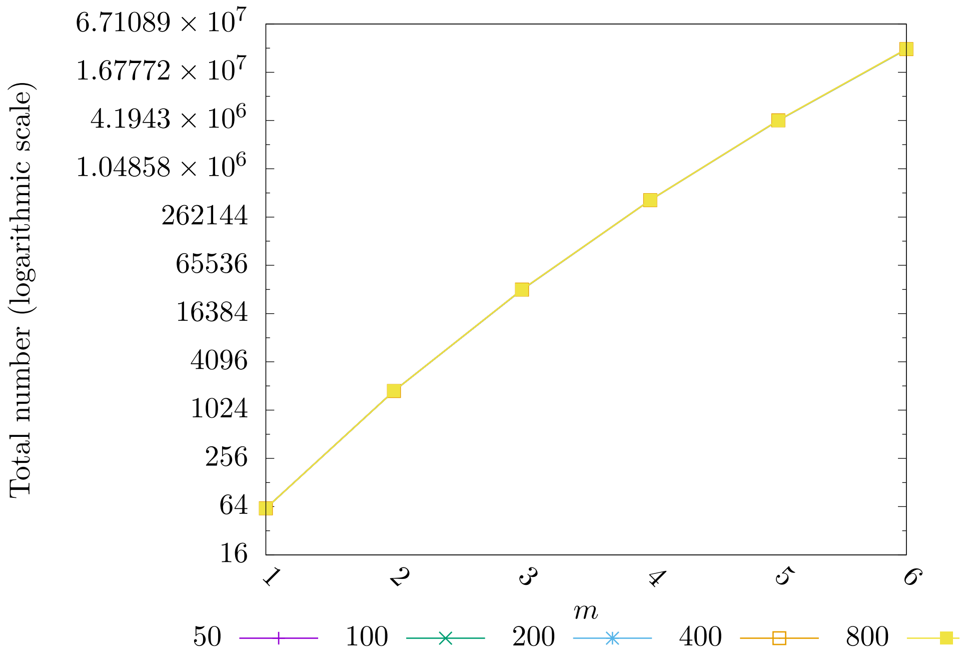


Figure A.3: Number of generated off-targets by the *DBG* method and $k = 6$.

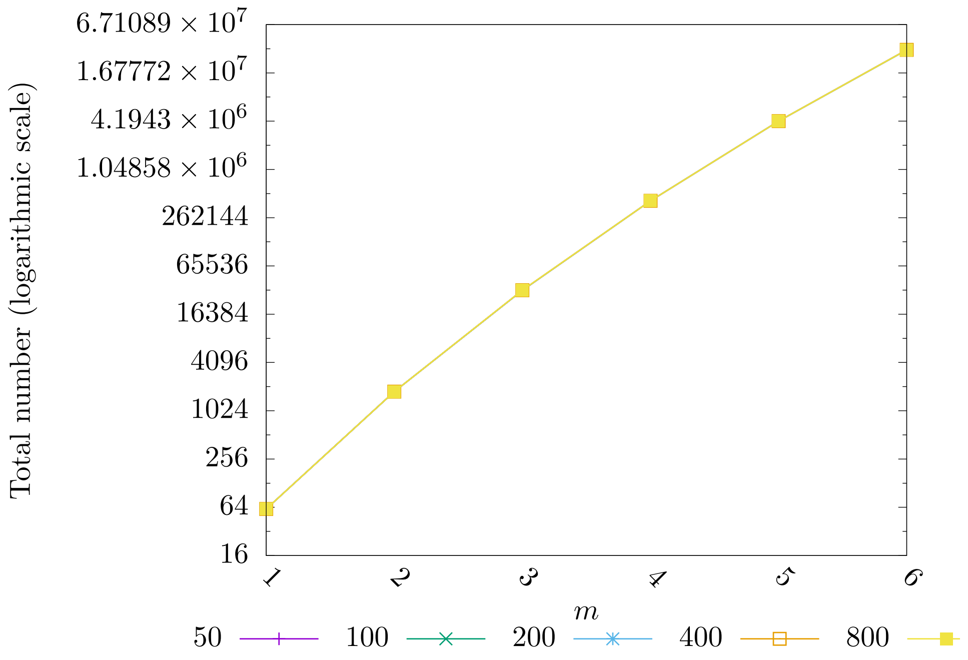


Figure A.4: Number of generated off-targets by the *DBG* method and $k = 7$.

Appendix B

Usage

B.1 Installation

The *SDSL* library needs to be installed on the system to be able to build the application. The *SDSL* library is located in code sources in the directory `src/crispr-tool/external/sdsl`. To build the *SDSL* a modern *C++* compiler that supports *C++11* and 64 bit operating system is required.

Installation is done by following this commands:

```
# cd src/crispr-tool/external/sdsl
# mkdir build
# cd build
# cmake ..
# make
# make install
```

The installation of the CRISPR tool is similar:

```
# cd src/crispr-tool
# mkdir build
# cd build
# cmake ..
# make crispr
```

This will build the application into the `exe/` directory. All possible executable that can be build are shown in next table.

crispr is a main application that runs scoring of the input guide on specified input using both *simple* and *DBG* generator.

benchmark_dbg creates the *DBG* from the input text.

benchmark_sa runs *SA* construction for tested algorithms.

debruijn_test was used for testing the *DBG* as it supports continuous input.

rawer is used to remove comment lines and new line symbols from FASTA files. It also currently removes all symbols that are not `A`, `C`, `G`, `T`, `N`.

B.2 Usage

CRISPR tool interface is simple. It supports three parameters `-i`, `-b` and `-c`.

`-b INPUT_FILE` indicates to create index and *DBG* from the input file. Resulted index will be stored in the same directory with suffix `.index` and *DBG* stored with suffix `-k.dbg`. `k` is equal to the k specified in the configuration file.

`-i INPUT_FILE` if the index and *DBG* files are already created.

`-c` configuration file.

Note that the index and *DBG* for input files are already precreated in the input directory, so use only `-i` parameter. If we want to create index and *DBG* from the input, make sure, that the same directory as input file, also contains directory `sdsl` (used to store temporary files during building process).

Appendix C

Acronyms

DNA	Deoxyribonucleic acid
RNA	Ribonucleic acid
PAM	Protospacer adjacent motif
CRISPR	Clustered Regularly Interspaced Short Palindromic Repeats
EOF	End Of File symbol
OS	Operating System

Appendix **D**

Contents of enclosed CD

```
readme.txt ........................ the file with CD contents description
exe ..................................... the directory with executables
src ....................................... the directory of source codes
  crispr-tool ................................. implementation sources
    bin .............. the directory with bash script for measurements
    external ....................... the directory of external libraries
    input ............................... the directory with input files
    output ............. the directory with results of all measurements
    src ................................ the directory of source codes
    CMakeLists.txt ... the file for automative build of the application
  thesis .............. the directory of LaTeX source codes of the thesis
text ......................................... the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
  thesis.ps .............................. the thesis text in PS format
```