# On the maximum size of condensed sequences neighbourhoods under the Levenshtein distance

by

**France Paquet-Nadeau**

B.Sc. Honours in Mathematics with Minor in Music, University of Ottawa, 2015

Master's Project Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master's of Science

in the
Department of Mathematics
Faculty of Science


**SIMON FRASER UNIVERSITY**
**Summer 2017**

# Approval

**Name:** **France Paquet-Nadeau**

**Degree:** **Master's of Science (Mathematics)**

**Title:** ***On the maximum size of condensed sequences neighbourhoods under the Levenshtein distance***

**Examining Committee:**

**Cédric Chauve**
Senior Supervisor
Professor _____

**Marni Mishna**
Co-Supervisor
Associate Professor _____

**Date Defended:** August 11, 2017

# Abstract

This work is centered around the sequence alignment problem and the complexity of the algorithms to generate the condensed-neighbourhood of a word. The condensed-neighbourhood consist of words of the neighbourhood which do not contain a prefix that is also part of the neighbourhood. Given a maximal number of errors $d$ and the length of a word $k$, we can determine an upper-bound on the number of words in the condensed-neighbourhood. We study the method from Gene Myers in [2, 1] to compute a bound using the edit-scripts model. We are able to derive a tighter bound using asymptotic analysis of the recurrences he introduced. We also study the method by Hélène Touzet [3] which uses finite deterministic automaton to count exactly the number of words in a $d$-neighbourhood. We used Python and Maple to implement the recurrences of both approaches, which allowed for comparison analysis. This research will help develop more efficient algorithms for sequence alignment, which is useful for genomics and error correction.

**Keywords:** sequence alignments, condensed-neighbourhood, algorithms for sequence alignments, deterministic finite automaton, Levenshtein distance.

# Acknowledgements

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In Bioinformatics, the analysis of large-scale sequence alignment algorithms relies on the properties of the size of the neighbourhood of a word. Given a very long text $T$ of length $N$ and a query string $W$ of length $k$, both on an alphabet $\Sigma$ of length $s$, we want to identify in $T$ the strings which are at distance at most $d$ from $W$. Words in a neighbourhood have a certain amount of differences from the original word we are searching. For example if we were searching for the word *abba* and wanted to search for words with 1 difference, then *abbb* would be one possibility as well as *aba*. These are only two examples of words in the 1-neighbourhood of *abba*, but there are many more. Knowing exactly how many words are in this neighbourhood is important to construct efficient search algorithms. The longer the initial word is, and as we allow more differences, the size of the neighbourhood becomes quite large. Instead of storing all the words of the neighbourhood, we can look for those words that share a prefix and store only that prefix. We keep only the prefixes which are words of the neighbourhood. In the last example with initial word *abba*, we could store *abb*, which is a prefix to *abbb, abbaa, abbab, abba* and is itself in the 1-neighbourhood of *abba*. Hence we are storing one word instead of five.

The advantage of searching for words in the condensed-neighbourhood of $W$ instead of searching for $W$ itself is as follows: suppose $W$ has length 100 and we are allowed nine differences in the original text $T$. Partitioning $W$ into 10 strings of length 10 means, by the Pigeon Hole principle, that at least one of those 10 strings will match exactly in $T$. Because we have to place 9 differences into 10 possible strings, at least one of the strings will not contain any differences. On the other hand, if we partition $W$ into 5 strings of length 20, then again by the Pigeon Hole principle, a string in the 1-neighbourhood of those 5 strings will match exactly in $T$, because of the five strings at least one will contain at most one difference. Instead of generating all the strings of the 1-neighbourhood, generating the strings of the condensed-neighbourhood results in a smaller number of strings to generate as well as overall shorter ones, allowing for a smaller memory use.

The aim of this project is to understand the algorithm presented by Gene Myers in [2], that uses recurrences on alignments to compute an upper-bound for the size of the condensed-neighbourhood of an initial word $W$ of length $k$. We will work with an **alphabet**, denoted $\Sigma$, which is a set of characters used in the text $T$ as well as the characters used to build $W$ and its neighbourhood strings. A **sequence**, or **word**, is a list of characters from the alphabet. A **prefix** is a list of characters at the beginning of a word. We are interested in finding distinct words that share a prefix in order to reduce as much as possible to number of strings we need to store. For example, the words *abaaba* and *abaac* share the prefix *abaa*. By storing only *abaa*, we store the necessary information to recover the two strings *abaaba* and *abaac*.

An **alignment** $S$ is defined over $\Sigma' = (\Sigma \cup \{-\})^2$. The functions $\pi_1$ and $\pi_2$ are sequences over $\Sigma$ obtained by taking respectively the upper and lower part of $S$ and removing the "-" characters. There are four possible operations to create an alignment. A **match** is a pair of letters $(x, y)$ form $\Sigma'$, such that $x = y \neq -$. A **substitution** is a pair of letters $(x, y)$ from $\Sigma'$ such that $x \neq y$, $x \neq -$ and $y \neq -$. An **insertion** is a pair of characters $(-, y)$ from $\Sigma'$ such that $y \neq -$. Finally a **deletion** is a pair of characters $(x, -)$ from $\Sigma'$ such that $x \neq -$. A sequence of these operations is an alignments. An **edit-script** is defined over $\Sigma \cup \{-\}$ and corresponds to the lower part of an alignment $S$. On the other hand, $\pi_2(S)$ corresponds to the word generated by the alignment. The **edit score** of an alignment $S$ is the total number of substitution, insertions and deletions in $S$.

We use the **Levenshtein distance** as our metric norm. It gives the minimum number of operations (insertions, deletions and substitutions) necessary to transform a word $W$ into a word $W'$.

For example, the following alignment,

| | | | | |
|---|---|---|---|---|
| a | b | a | a | - |
| b | b | - | c | a |

generates the edit-script *bb-ca* and the word *bbca*. It starts with a substitution, followed by a match, a deletion, a substitution and ends with an insertion. If *abaa* is our original word, then $Lev(abaa, bbca) = 2$, because we can create the word *bbca* with this optimal alignment instead:

| | | | |
|---|---|---|---|
| a | b | a | a |
| b | b | c | a |

.

The **d-neighbourhood** of $W$ consists of all the words that are at a distance at most $d$ from $W$. We denote by $N_d(W) = \{v \mid Lev(v, w) \leq d\}$ the $d$-neighbourhood of $W$. For example, if $\Sigma = \{a, b\}$ and $W = baa$, then $N_1(baa) = \{aa, ba, abaa, bbaa, baaa, baba, baab, aaa, bba, bab, baa\}$. Notice how *baa* is included in the list, because $Lev(baa, baa) = 0 < 1$. Similarly, the **condensed-neighbourhood** of a word $W$ consists of the words in the $d$-neighbourhood

that do not have a prefix in the neighbourhood. We denote the condensed-neighbourhood as $\overline{N_d}(W) = \{v \mid v \in N_d(W)$ and no prefix of $v$ is in $N_d(W)\}$. In this example, $\overline{N_1}(baa) = \{aa, abaa, ba, bba\}$. We can easily verify that every word in $N_1(baa)$ starts by one of the four strings in $\overline{N_1}(baa)$. We also have $\overline{N_d}(k) = \max_{W,|W|=k}\{|\overline{N_d}(W)|\}$, the maximum size of the condensed-neighbourhood on all words of length $k$.

In Chapter 2 we focus on the algorithm of Myers presented in [2, 1] to generate an upper-bound on the size of the condensed-neighbourhood. We explain in details the recurrences and how they generate alignments from the four possible operations. In Chapter 3 we study the upper-bounds on the size of the condensed-neighbourhood. From the recurrences in [2], we derive a new upper-bound for the size of the condensed-neighbourhood by using generating functions and asymptotic analysis. We apply the method to the first version of the recurrences in [1] and then compare these two asymptotic bounds for different alphabet sizes.

In Chapter 4 we will focus on the method of Touzet [3], which uses finite deterministic automaton to count exactly the number of words in a $d$-neighbourhood. This method differs from the previous one by Myers from the fact that it does not uses alignments and also does not mention the condensed-neighoburhood. Instead it transforms words into bit vectors which are used as transitions in deterministic automata. Counting the number of words accepted by the final automaton gives the exact number of words in the $d$-neighbourhood of $W$. Finally in Chapter 5 we present a comparison analysis of both methods. After implementing both methods in Python, we were able to generate data on the exact number of words in the $d$-neighbourhood of $W$ as well as the upper-bound on the condensed-neighbourhood. By adapting the recurrences in [2] we were able to generate the words of the condensed-neighbourhood as well the number of words that are generated multiple times, and hence evaluate the redundancy of the recurrences.

# Chapter 2

# On "What's behind BLAST", by Gene Myers

In specific applications, such as searching a DNA or protein database, the search text $T$ can have length $N$ in the order of millions or billions. From an initial word $W$, the goal is to identify in $T$ the occurrences corresponding to words at a distance at most $d$ to $W$. In [2], Myers proposed an algorithm that uses the idea of a condensed-neighbourhood to search for these occurrences of $W$ in $T$. Assuming $T$ has been indexed beforehand, making the information about occurrences of $W$ easily accessible, and assuming $d$ is relatively small, it is more efficient to generate the condensed-neighbourhood of $W$ and search for those words in $T$, rather than search for exact matches of $W$ in $T$.

The aim of Myers in [2] is to give an algorithm to compute an upper-bound on the maximal size of the condensed-neighbourhood of words of length $k$ in which we introduce at most $d$ differences. In this chapter we will focus on the recurrences given in [2] to compute this bound. It is known that this bound is not optimal, because it generates some words multiple times as well as it generates words from the $d$-neighbourhood rather than from the condensed-neighbourhood. Understanding exactly the words generated by these recurrences would allow for improvement.

Using the four operations to create alignments: insertion, deletion, substitution and match, the algorithm defines rules to avoid some redundancy in the words generated.

## 2.1 The Recurrences

The goal of the recurrence relations presented in [2] is to give an upper-bound on the maximal size of the condensed-neighbourhood of all words $W$ of length $k$. We are working with a text $T$ of length $N$, where we consider the last $k$ symbols of it.

text: n n-1 ... k+1 k k-1 k-2 ... 2 1

We perform induction on the characters of the text, going from left to right. In the remaining $k$ symbols, we want to introduce at most $d$ differences. In order to get an accurate bound, we need to consider some restrictions on the alignments. Notice the following redundant combinations:

1. a deletion followed by an insertion has the same effect on the edit-script as a substitution. For example having $\begin{array}{|c|c|}\hline a & - \\ \hline - & b \\ \hline \end{array}$ or $\begin{array}{|c|}\hline a \\ \hline b \\ \hline \end{array}$ in an alignment will have the effect of a substitution on the edit-script. However doing a deletion followed by an insertion requires two operations whereas a substitution only requires one. Therefore we choose to use substitution and not allow the combination deletion + insertion.

2. A deletion followed by a substitution generates the same string as a substitution followed by a deletion. In this case, the difference is the order in which we perform the operations. In order to avoid some redundancy, we chose to allow only substitution followed by deletion and we reject the possibility of deletion followed by substitution.

3. An insertion followed by a substitution generates the same script as a substitution followed by an insertion. Once again the only difference here is the order in which we perform both operation as they both give a score of 2. We choose to allow only substitution followed by insertion.

4. An insertion followed by a deletion is the same as either a match or a substitution, depending if the symbol inserted is the same as the one deleted or not. In this case because a match has a score of 0 and an insertion a score of 1 compared to a score of 2 for an insertion followed by a deletion, it is natural to choose the match or the substitution. We therefore will not allow an insertion to be followed by a deletion.

These restrictions will help define which combinations to consider for the recurrences. Because the recurrences count the number of edit-scripts with as little redundancy as possible and these restrictions reduce the redundancy of the words generated. We will denote by $S_2(k, d)$ the size of the set of alignments generated by the recurrences, which contain $d$ differences in the last $k$ symbols of the query. Each term of the recurrences can be represented with a table of the symbols left in the query. The first line of the table shows the number of symbols left in the query, the second line shows the characters from the text and the third line represents the operations performed. Lines two and three together correspond to the alignment.

- A match of symbol $k$ is represented by $S_2(k-1, d)$; there are $k-1$ remaining symbols to introduce $d$ differences. Symbol $k$ is a match.

| k | k-1 | ... |
|---|---|---|
| x | ... | ... |
| x | ... | ... |

- A substitution of symbol $k$ is represented by $(s-1)S_2(k-1,d-1)$; there are $s-1$ non-redundant symbols available for the substitution and there are $k-1$ symbols left to introduce $d-1$ differences.

| k | k-1 | ... |
|---|---|---|
| x | ... | ... |
| y | ... | ... |

- A match followed by a sequence of insertions that ends in a match, $(s-1)\sum_{j=0}^{d-1} s^j S_2(k-2,d-1-j)$; symbol $k$ is a match, the first insertion has $s-1$ non-redundant choices of symbols and the following $j$ insertions have $s$ choices of symbols. Because a sequence of insertions terminates with a match, symbol $k-1$ will be a match and we are left with $k-2$ symbols to introduce $d-1-j$ differences.

| k | - | - | k-1 | k-2 | ... |
|---|---|---|---|---|---|
| x | - | - | x | ... | ... |
| x | y | $j>0$ inserts | x | ... | ... |

- A substitution followed by a sequence of insertions that ends in a match, $(s-1)^2\sum_{j=0}^{d-2} s^j S_2(k-2,d-2-j)$; symbol $k$ has $s-1$ choices of non-redundant substitution and the first insertion also has $s-1$ choices of characters, hence $(s-1)^2$. The following $j$ insertions have no restrictions and again the insertion sequence terminates with a match on symbol $k-1$. We are left with $k-2$ symbols to introduce $d-1-j$ differences.

| k | - | - | k-1 | k-2 | ... |
|---|---|---|---|---|---|
| x | - | - | x | ... | ... |
| y | x | $j>0$ inserts | x | ... | ... |

- A sequence of deletions starting at symbol $k$ that ends in a match, $\sum_{j=0}^{d-1} S_2(k-2-j,d-1-j)$; symbol $k$ is deleted as well as the following $j$ symbols. A sequence deletions also terminates with a match, hence we are left with $k-1-j$ symbols to introduce $d-1-j$ differences.

| k | k-1 ... k-j | k-1-j | k-2-j | ... |
|---|---|---|---|---|
| x | x ... x | x | ... | ... |
| - | $j>0$ deletions | x | ... | ... |

- Insertions before the first symbol of the query, $\sum_{j=1}^{d} s^j S_2(k-1, d-j)$; each insertions as a choice of $s$ characters and again the sequence as to end with a match, this case on symbol $k$. We are left with $k-1$ symbols to introduce $d-j$ differences.

| - | k | k-1 | ... |
|---|---|---|---|
| - | x | ... | ... |
| $j$ inserts | x | ... | ... |

Because we want to generate only the condensed-neighbourhood, the stopping conditions are $k \leq d$ or $d = 0$. If we wanted to generate the full $d$-neighbourhood, then we would need to stop when $d = 0$. Since we only want to condensed-neighbourhood, we can reduce the computations by allowing to stop when $k \leq d$. Because the condensed-neighbourhood consist of words that do not share a prefix, it is enough to stop when $k \leq d$. If we reach $k \leq d$ symbols in the query, assuming $d > 0$, then we haven't used all the differences in the edit-script and also that there are more differences left to insert in the string that there are characters. Hence all the strings generated when $k > d$ contain a prefix in the neighbourhood.

If we reach $d = 0$, then the remaining symbols can only be matches and the word generated is at distance exactly $d$ from the original word. The only combinations not allowed by these recurrences for the condensed-neighbourhood but that could be possible to generate the complete neighbourhood are: sequences of insertions separated by a single match, a sequence of insertion separated from a sequence of deletions by a substitution and a sequence of deletions at the end of a word which doesn't end with a match. We do not need to consider these case for the condensed-neighbourhood because they will contain a prefix that is generated by the existing recurrences.

If we reach $d = 0$, then we've created $d$ differences within the $k$ symbols or less. If we didn't use all $k$ symbols to introduce the $d$ differences, we complete the word by having matches on the remaining symbols. The word created will therefore be in the neighbourhood.

If instead we reach $k \leq d$, then the remaining symbols of the query can all count as differences. But the string created is in the neighbourhood since it contains at most $d$ differences from the original string. This happens if we reach $k = d$, otherwise if we have $k < d$, then the string created will contain less than $d$ differences from the original sting. Hence it will be prefix of all the other strings with the extra differences.

**Lemma 1.**

$$S_2(k,d) = \begin{cases} 1 & \text{if } k \leq d \text{ or } d = 0 \\ S_2(k-1,d) + (s-1)S_2(k-1,d-1) & \text{otherwise,} \\ +(s-1)\sum_{j=0}^{d-1} s^j S_2(k-2,d-1-j) & \\ +(s-1)^2 \sum_{j=0}^{d-2} s^j S_2(k-2,d-2-j) & \\ +\sum_{j=0}^{d-1} S_2(k-2-j,d-1-j) & \end{cases}$$

$$\overline{N_d}(k) \leq S_2(k,d) + \sum_{j=1}^{d} s^j S_2(k-1,d-j)$$

## 2.2 Proof of the recurrences

As previously mentioned, the possible operations to create edit-scripts are *match, insertion, deletion* and *substitution*. Because the goal of these recurrences is to reduce the redundancy in the edit-scripts generated, we do not want to allow every combinations of the four operations *match, insertion, deletion* and *substitution*. Based on the observations of redundant combinations in section 2.1, we choose to only allow the following combinations to create edit sequences:

1. a match can be followed by anything;
2. a substitution can be followed by anything;
3. an insertion can be followed only by another insertion or a match;
4. a deletion can be followed only by another deletion or a match.

Hence $S_2(k,d)$ needs to take these restrictions into account. So $S_2(k,d)$ will have (at least) four components, one for each case.

- First, (1.) can be written as $S_2(k-1,d)$, meaning symbol $k$ is a match, leaving $k-1$ symbols to introduce $d$ differences.
- (2.) can be divided into two parts. With the same idea, we have $(s-1)S_2(k-1,d-1)$, meaning symbol $k$ is a substitution. We have $s-1$ possibilities for non-redundant substitution (on an alphabet of size $s$), leaving $k-1$ symbols to introduce now $d-1$ differences.

  Because we have more restrictions on insertions and deletions, we introduce $I(k,d)$ and $D(k,d)$ to represent these, respectively.

- The other case is when we have a substitution followed by an insertion. This leads to $(s-1)^2 I(k-1, d-2)$, so (3.) can be written as $(s-1)I(k-1, d-1)$, meaning we insert directly after symbol $k$. The term $(s-1)$ comes from the fact that we do not want to insert the same symbol as the one preceding it (in this case symbol $k$) because it is not possible to distinguish between insertion before or after that symbol.
- Finally, we can write (4.) as $D(k-1, d-1)$, meaning we delete symbol $k$, leaving $k-1$ symbols to introduce $d-1$ differences.

We therefore obtain the following equation:

$$S_2(k,d) = S_2(k-1,d)+(s-1)(S_2(k-1,d-1)+I(k-1,d-1))+(s-1)^2 I(k-1,d-2)+D(k-1,d-1) \tag{2.1}$$

We would like to have everything in terms of $S_2$, so we concentrate on solving $I(k,d)$ and $D(k,d)$ in terms of $S_2$. Formally, $I(k,d)$ represents the sequences of terms that start with an insertion whereas $D(k,d)$ represents the sequences of terms that start with a deletion. In the case of $I(k,d)$, we start the sequence of at least one insertions after symbol $k+1$ and will have a match on symbol $k$. For $D(K,d)$, we have a sequence of $j \geq 1$ deletions starting on symbol $k+1$ and ending with a match on symbol $k-j+1$.

### 2.2.1 Solving *I(k,d)*

Following the restrictions of case (3.), there are only two possible operations once we do an insertion. Either do another insertion or end with a match. This can be written as:

$$I(k,d) = sI(k,d-1) + S_2(k-1,d)$$

| ... | k+1 | - | k | k-1 | ... | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | | insert | $\vdash$ | | $d$ differences | | $\dashv$ |

$=$

| ... | k+1 | - | - | k | k-1 | ... | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| | | insert | insert | $\vdash$ | | $d-1$ differences | | $\dashv$ |

$+$

| ... | k+1 | - | k | k-1 | k-2 | ... | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| | | insert | match | $\vdash$ | | $d$ differences | | $\dashv$ |

We deduce that $I(k-1, d-1) = sI(k-1, d-2) + S_2(k-2, d-1)$.

| ... | k | - | - | k-1 | k-2 | ... | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| | | insert | insert | $\vdash$ | | $d-2$ differences | | $\dashv$ |

| ... | k | - | k-1 | k-2 | k-3 | ... | 2 | 1 |
|---|---|---|---|---|---|---|---|---|
| | | insert | match | $\vdash$ | | $d-1$ differences | | $\dashv$ |

We can keep going:

$$\begin{aligned}
I(k-1,d-2) &= sI(k-1,d-3) + S_2(k-2,d-2)\\
I(k-1,d-3) &= sI(k-1,d-4) + S_2(k-2,d-3)\\
&\dots\\
I(k-1,2) &= sI(k-1,1) + S_2(k-2,2)\\
I(k-1,1) &= sI(k-1,0) + S_2(k-2,1)\\
I(k-1,0) &= S_2(k-2,0)
\end{aligned}$$

Now we can substitute back:

$$\begin{aligned}
I(k-1,1) &= sS_2(k-2,0) + S_2(k-2,1)\\
I(k-1,2) &= s(sS_2(k-2,0) + S_2(k-2,1)) + S_2(k-1,2)\\
&= s^2S_2(k-2,0) + sS_2(k-2,1) + S_2(k-2,2)\\
&\dots\\
I(k-1,d-1) &= s^{d-1}S_2(k-2,0) + s^{d-2}S_2(k-2,1) + \dots + sS_2(k-2,d-2) + S_2(k-2,d-1)\\
&= \sum_{j=0}^{d-1} s^j S_2(k-2,d-1-j)
\end{aligned}$$

Similarly, we get that $I(k-1,d-2) = \sum_{j=0}^{d-2} s^j S_2(k-2,d-2-j)$.

### 2.2.2 Solving *D(k,d)*

Just like insertions, deletions can only be followed by an other deletion or a match. $D(k,d)$ counts the number of $d$-edit scripts that start with a deletion of symbol $k+1$. We have:

$$D(k,d) = D(k-1,d-1) + S_2(k-1,d)$$

| ... | k+1 | k | k-1 | ... | 2 | 1 |
|---|---|---|---|---|---|---|
| | delete | $\vdash$ | | $d$ differences | | $\dashv$ |

$=$

| ... | k+1 | k | k-1 | ... | 2 | 1 |
|---|---|---|---|---|---|---|
| | delete | delete | $\vdash$ | $d-1$ differences | | $\dashv$ |

$+$

| ... | k+1 | k | k-1 | k-2 | ... | 2 | 1 |
|---|---|---|---|---|---|---|---|
| | delete | match | $\vdash$ | | $d$ differences | | $\dashv$ |

Similarly, we can solve:

$$\begin{aligned}
D(k-1,d-1) &= D(k-2,d-2) + S_2(k-2,d-1)\\
D(k-2,d-2) &= D(k-3,d-3) + S_2(k-3,d-2)\\
D(k-3,d-3) &= D(k-4,d-4) + S_2(k-4,d-3)\\
&\dots\\
D(k-(d-2),2) &= D(k-(d-1),1) + S_2(k-(d-2),2)\\
D(k-(d-1),1) &= D(k-d,0) + S_2(k-d,1)\\
D(k-d,0) &= S_2(k-d-1,0)
\end{aligned}$$

We can now substitute back:

$$\begin{aligned}
D(k-(d-1),1) &= S_2(k-d-1,0) + S_2(k-d,1)\\
D(k-(d-2),2) &= S_2(k-d-1,0) + S_2(k-d,1) + S_2(k-(d-2),2)\\
&\dots\\
D(k-2,d-2) &= S_2(k-d-1,0) + S_2(k-d,1) + \dots + S_2(k-4,d-3) + S_2(k-3,d-2)\\
D(k-1,d-1) &= S_2(k-d-1,0) + S_2(k-d,1) + \dots + S_2(k-3,d-2) + S_2(k-2,d-1)\\
&= \sum_{j=0}^{d-1} S_2(k-2-j,d-1-j)
\end{aligned}$$

This hence gives the result we were expecting.

### 2.2.3 Back to the recurrences

Substituting back the results of $I(k,d)$ and $D(k,d)$ in 2.1 gives the statement of Lemma 1. We now understand each term and how to obtain them from the intuition of the rules on the edits.

## 2.3 Earlier work

The recurrences of chapter 2.1 were in fact the second published version by Myers. In his earlier work [1], he gave his first presentation of his idea for an algorithm for keyword search. In this paper his version of the recurrences were simpler and did not consider as many restrictions on the combination of operations. Because we will want to compare the two version of the recurrences, we present the first version here. To be consistent with [1], we will denote by $Z(k,d) = \max\{|\overline{N_d}(W)|\text{: } W \text{ is a word of length } k\}$ what we have already defined as $\overline{N_d}(k)$. Hence the first recurrences will bound $Z(k,d)$ whereas the improved version of 2.1 bound $\overline{N_d}(k)$. Even though they have the same definition, the fact

that recurrences for $Z(k,d)$ allow more redundancy means that the bounds obtain by both $Z(k,d)$ and $\overline{N_d}(k)$ are quite different.

In this early version, some possibilities for redundant edit-scripts are mentioned. It is noted that doing a deletion followed by an insertion will produce the same edit-script as doing a substitution. Because we are aware of these equivalent edit-script, we do not want to perform every combination or our four operations *match, insertion, deletion* and *substitution*. Here is a list of the possible operation we can perform on each of the symbols of the query. We again build the edit-scripts by considering each symbol of the query, starting on the left and moving one symbol to the right with every operation performed. At each symbol of the query we can perform one of the four operations, following these rules:

1. insertions before the first symbol of the query;
2. a match;
3. a deletion;
4. insertions after a symbol of the query;
5. a substitution for a different character of the alphabet followed by zero or more insertions.

By considering only restrictions 1-5, we obtain an upper-bound on $Z(k,d)$, the maximal cardinality of the condensed-neihbourhood of words of length $k$.

**Lemma 2.**

$$S_1(k,d) = \begin{cases} 1 & \text{if } d = 0, \\ 2s & \text{if } d = 1 \text{ and } k = 1, \\ (2s-1)|\Sigma|^{d-1} & \text{if } d > 1 \text{ and } k = 1, \\ S_1(k-1,d) + S_1(k-1,d-1) & \text{otherwise,} \\ +(2s-1)\sum_{j=1}^{d} s^{j-1} S_1(k-1,d-j) & \end{cases}$$

$Z(k,d) \leq \sum_{j=0}^{D} s^j S_1(k,d-j)$

*Proof.* This proof is very similar to the one of Lemma 1. We explain each terms separately.

- $S_1(k,0) = 1$ because there is only one empty edit-script.
- When $d = k = 1$, we have only one symbol to perform 1 operation, either a deletion, substitution or insertion. There is only one script possible for the deletion, $(s-1)$ possible scripts for the substitution and $s$ scripts for an insertion. In total that makes $1 + (s-1) + s = 2s$ possible edit-scripts.

- In the case $d > 1$ and $k = 1$, we have one symbol left to perform more than 1 operations. Since we cannot do a deletion, the possibilities are either $d$ insertions or 1 substitution followed by $d-1$ insertions. There are $s^d$ possible edit-scripts for the $d$ insertions and $(s-1)s^{d-1}$ possible scripts for the second case. Together there are $(2s-1)s^{d-1}$ possible edit-scripts for that case.

- For the other cases where $d > 1$ and $k > 1$ we have:

  1. the term $S_1(k-1, d)$ correspond to having a match on the first symbol;
  2. $S_1(k-1, d-1)$ corresponds to the deletion of the first symbol;
  3. inserting $j \in [1, d]$ symbols after the first symbol and $d-j$ edits on the remaining $k-1$ symbols for a total of $s^j S_1(k-1, d-j)$ scripts or substitute the first symbol and insert $j-1$ symbols, $j \in [1, d]$, afterwards and completing with $d-j$ edits in the remaining $k-1$ symbols, for a total of $(s-1)s^{j-1}S_1(k-1, d-j)$ scripts. Since we make at least one insertion in the first case, we obtain the term $(2s-1)\sum_{j=1}^{d} s^{j-1}S_1(k-1, d-j)$.

- Finally to obtain the bound on $Z(k, d)$ we need to consider the scripts with insertions before the first symbol. There are $s^j S_1(k, d-j)$ possibilities, as there are no restrictions on those insertions. Hence the bound is $Z(k, d) \leq \sum_{j=0}^{D} s^j S_1(k, d-j)$.

□

Although noticing some redundant combinations of operations, we notice that the recurrences $S_1(k, d)$ do not actually forbid them from being generated. Indeed, it is mentioned that an insertion followed by a deletion is equivalent to a match or a substitution, but in the recurrences there are actually no restriction on the operation we can perform after an insertion. Now that we have both versions, we will be able to compare them.

## 2.4 Which words do these recurrences generate?

Since the goal of these recurrences is to generate condensed-neighbourhoods with minimal redundancy, we studied the words generated to gain a better understanding of what exactly these recurrences generate. We were able to implement the recurrences in Python and adapt them to generate the words obtained from the alignments. Knowing the recurrences are are not optimal for the size of the condensed-neighbourhood, we want to understand exactly which words they generate. This would help to either improve the recurrences by reducing the redundancy or to modify them to obtain an upper-bound on the size of the $d$-neighbourhood.

### 2.4.1 Redundant words

First, it appears that some of the words are generate multiple times. For example, when generating the words of the condensed-neighbourhood of $aaaaa$ with $d = 2$, the word $aaa$ is generated 10 times. This is due to the fact that deleting any 2 letters of the original word will have the same effect on the alignment and generate the word $aaa$. An other example is the word $baaa$ which is generated 4 times. In this case we have to substitute the first letter to a $b$, but again deleting any of the remaining letter generates the same word.

For now it seems difficult to identify a new restriction on the alignments that would reduce the redundancy for every initial word of length $k$. Within all the words of a same length, some initial words will have multiple words with low redundancy, words that are generated between 2 and 4 times, and some initial words will have less redundant words but these words will be generated 10 times or more. This make it difficult to identify new restrictions on the alignments because at first glance there is not one rule that generates the redundancy or that is missing to reduce the redundancy.

### 2.4.2 Words from the neighbourhood

When analyzing the number of words generated by the recurrences, we notice that the upper-bound is not a good approximation of the real value. This is not only the cause of the redundant words, but also the fact that in the condensed-neighbourhood we need only generate the prefixes. What happens here is a significant amount of words generated contain a prefix which is also in the neighbourhood. This leads to those words not being needed in the count for the condensed-neighbourhood, meaning there might be a problem with the stopping conditions of the recurrences.

# Chapter 3

# Obtaining a better upper-bound on the recurrences

In his first published version of his algorithm for keyword search [1], Myers also presented an upper-bound function on his recurrences. He introduces the function $B(k, d, c)$ which is an upper-bound on the recurrences $S_1(k, d)$ and also gives an upper-bound on $Z(k, d)$ when multiplied with the appropriate factor. This function $B(k, d, c)$ also appears in [2] as an upper-bound on $S_2(k, d)$ and $\overline{N_d}(k)$.

In this chapter we first compare the function $B(k, d, c)$ with $S_2(k, d)$ and then proceed to find a better upper-bound by using generating functions. We present an asymptotic analysis of both recurrences $S_1(k, d)$ and $S_2(k, d)$, and how they compare to each other.

## 3.1 The original upper-bound

The function $B(k, d, c)$ is first introduced in [1], although it is not entirely clear how i was obtained. It is mentioned in [2] that using induction on the function $S_2(k, d)$ yields the function $B(k, d, c)$. It is defined by $B(k, d, c) = (\frac{c+1}{c-1})^k c^d s^d$ for any value $c \geq 1$. The bounds we obtain are the following:

$$S_2(k, d) \leq B(k, d, c) \quad \text{and} \quad \overline{N_d}(k) \leq \frac{c}{c-1} B(k, d, c) \tag{3.1}$$

It should be mentioned that for the value $c^\star = \epsilon^{-1} + \sqrt{1 + \epsilon^{-2}}$ where $\epsilon = \frac{d}{k}$, the function $B(k, d, c)$ is minimal and therefore the best possible bound is obtained. This value $c^\star$ is chosen because $\epsilon \in [0, 1]$, hence $c^\star \in [1 + \sqrt{2}, \infty]$ and therefore the value $1 + \sqrt{2}$ is the minimal possible value for $c$. Hence $\frac{c^\star}{c^\star - 1} = \frac{1+\sqrt{2}}{\sqrt{2}} \approx 1.7071$ and the value 1.708 was chosen to approximate it. With the minimal value $c^\star$, optimal bounds on the recurrences and the condensed-neighbourhood are now:

$$S_2(k,d) \leq B(k,d,c^\star) \quad \text{and} \quad \overline{N_d}(k) \leq 1.708 B(k,d,c^\star) \tag{3.2}$$

By plotting both functions, we are able to see how $B(k,d,c^\star)$ compares to $\overline{N_d}(k,d)$. We can see on figures 3.1, 3.2 and 3.3 that the function $B(k,d,c)$, even evaluated at $c^\star$ is far from optimal. Even when $c = c^\star$ is minimal, the function grows much faster than the one it bounds. These figures are for an alphabet size of 2.

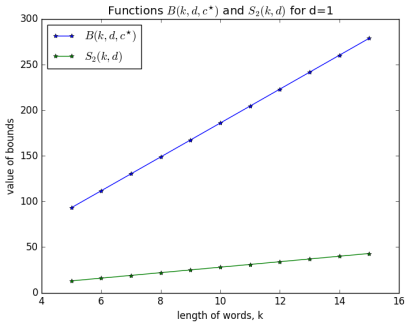


Figure 3.1: Comparison between $B(k,d,c^\star)$ and $S_2(k,d)$ for $d = 1$.

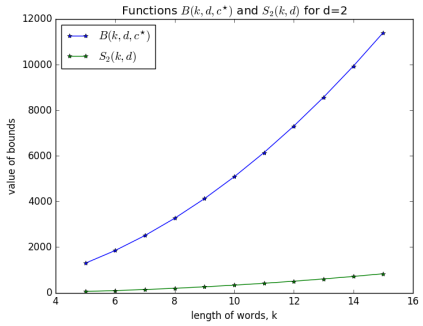


Figure 3.2: Comparison between $B(k,d,c^\star)$ and $S_2(k,d)$ for $d = 2$.

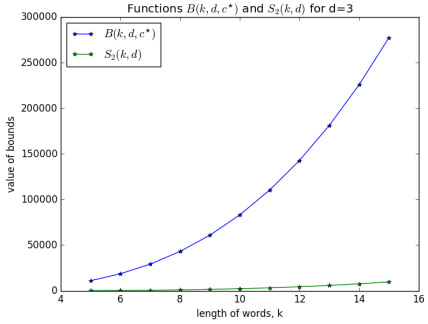


Figure 3.3: Comparison between $B(k,d,c^\star)$ and $S_2(k,d)$ for $d = 3$.

Given these observations, we wanted to obtain a tighter function to approximate $S_2(k,d)$. We initially believed that the data given by computing $S_2(k,d)$ generated an exponential function. By using curve fitting on the data we realized that the results grew according to a polynomial function of degree $d$, the number of errors allowed. With the experiments, we were confident that the new upper-bound function needed only be polynomial in $d$. In the original function $B(k,d,c)$, the term $\left(\frac{c+1}{c-1}\right)^k$ is not a power of $d$ and this could be why the approximation isn't as good as it could be. For a new upper-bound we would hopefully get rid of this term or replace it with an appropriate degree. From the curve fitting, the equation corresponding to the curves for $\overline{N_d}(k)$ are given by the equations $a_d x^d + a_{d-1}x^{d-1} + \cdots + a_1 x + a_0$, where $a_d, \ldots, a_0$ are constants.

Using generating functions and asymptotic analysis on the recurrences for $S_2(k,d)$, we were able to determine a new upper-bound function. We used Maple to encode the process of transforming the recurrences into generating function and then verified with the coefficients of the series the accuracy of the functions generated. In the next section we show how we were able to obtain a new upper-bound function from the recurrence $S_2(k,d)$.

## 3.2 A new bound for $S_2(k,d)$

To obtain a more accurate upper-bound function than $B(k,d,c)$ for the recurrences $S_2(k,d)$, we will transform those recurrences into generating functions. The resulting rational function generates a series in $z$ in which each coefficient of $z$ corresponds to the term generated by the recurrences. We will denote by $F_d(z)$ the rational generating function corresponding to the series and $F(k,d)$ will be the expression of the asymptotic function obtained.

To solve $F_d(z)$ we will keep $s$, the size of the alphabet, as a variable which we can easily substitute in the function $F_d(z)$. Starting with the recurrences for $S_2(k,d)$ as presented in Lemma 1, we know $S_2(k,0) = 1$, hence $F_0[z] = 1$. Therefore the first case we need to solve is for $d = 1$. From the recurrences of $S_2(k,d)$ we obtain the following,

$$\begin{aligned} S_2(k,1) &= S_2(k-1,1) + (s-1)S_2(k-1,0) + (s-1)S_2(k-2,0) + S_2(k-2,0) \\ &= S_2(k-1,1) + (s-1) + (s-1) + 1 \\ &= S_2(k-1,1) + 2s - 1. \end{aligned}$$

This expression is now what we want to transform. Converting to generating functions:

$$\begin{aligned} S_2(k,1) &= S_2(k,1) + 2s - 1 \\ \sum_{k=1}^{\infty} S_2(k,1)z^k &= \sum_{k=1}^{\infty} S_2(k-1,1)z^k + \sum_{k=1}^{\infty}(2s-1)z^k \\ \sum_{k=1}^{\infty} S_2(k,1)z^k &= z\sum_{k=1}^{\infty} S_2(k-1,1)z^{k-1} + (2s-1)\sum_{k=1}^{\infty} z^k \\ F_1(z) - 1 &= zF_1(z) + \frac{(2s-1)z}{1-z} \\ F_1(z)(1-z) &= \frac{(2s-1)z}{1-z} + 1 \\ F_1(z) &= \frac{(2s-1)z + 1 - z}{(1-z)^2} \\ F_1(z) &= \frac{(2s-2)z + 1}{(z-1)^2} \end{aligned}$$

Evaluating for $s = 2$ we obtain $F_1(z) = \frac{2z+1}{(1-z)^2}$, which gives the same coefficients as $S_2(k, d)$. Repeating this process we are able to solve $F_d(z)$ for any $d > 1$. In general, we find that the coefficients of $F_d(z)$ follow the asymptotical growth of $\frac{(2s-1)^d k^d}{d!}$.

As we can see on figures 3.4, 3.5, 3.6 and 3.7, the asymptotic function $F(k, d) = \frac{(2s-1)^d k^d}{d!}$ gives a much better approximation of $S_2(k, d)$ than $B(k, d, c^\star)$ for an alphabet of size 2.

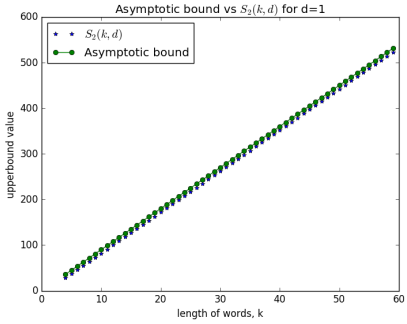


Figure 3.4: Comparison between the new asymptotic bound and $S_2(k, d)$ for $d = 1$.

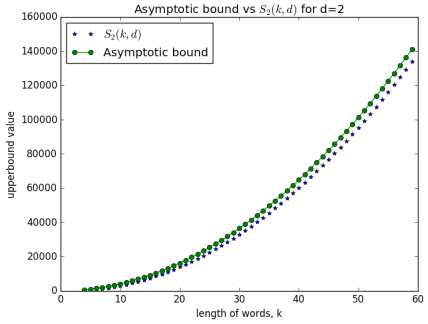


Figure 3.5: Comparison between the new asymptotic bound and $S_2(k, d)$ for $d = 2$.

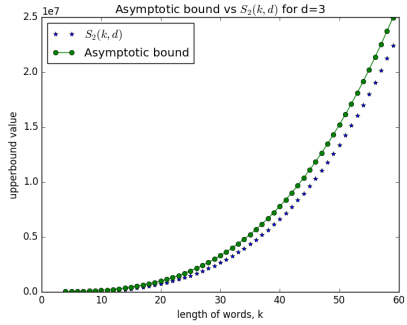


Figure 3.6: Comparison between the new asymptotic bound and $S_2(k, d)$ for $d = 3$.

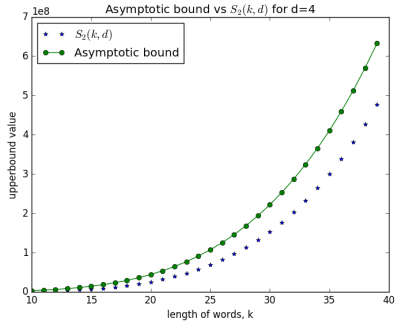


Figure 3.7: Comparison between the new asymptotic bound and $S_2(k, d)$ for $d = 4$.

## 3.3 Extending to $\overline{N_d}(k)$

Now that we have transformed the recurrences for $S_2(k, d)$ into generating functions and obtained the new function $F(k, d)$, we can easily apply the same method to obtain a bound on $\overline{N_d}(k)$. In [2] the bound on the condensed-neighbourhood is given by

$$\overline{N_d}(k) \leq S_2(k, d) + \sum_{j=1}^{d} s^j S_2(k-1, d-j). \tag{3.3}$$

Here we will denote by $Z_d(z)$ the generating function corresponding to the bound on $\overline{N_d}(k)$, namely $Z_d(z) = \sum_{k=0}^{\infty} S_2(k,d)z^k + \sum_{k=0}^{\infty}\sum_{j=1}^{d} s^j S_2(k-1,d-j)z^k$. Hence $Z_d(z)$ will give an upper-bound on $\overline{N_d}(k)$.

Applying the same technique as previously yield the following:

$$\begin{aligned}
Z_d(z) &= \sum_{k=0}^{\infty} S_2(k,d)z^k + \sum_{k=0}^{\infty}\sum_{j=1}^{d} s^j S_2(k-1,d-j)z^k \\
&= F_d(z) + z\sum_{k=0}^{\infty}\sum_{j=1}^{d-1} s^j S_2(k-1,d-j)z^{k-1} + s^d\sum_{k=0}^{\infty} z^k \\
&= F_d(z) + z\sum_{k=1}^{\infty}\sum_{j=1}^{d-1} s^j [S_2(k-1,d-j)z^{k-1} - 1] + s^d\sum_{k=0}^{\infty} z^k \\
&= F_d(z) + z\sum_{k'=0}^{\infty}\sum_{j=1}^{d} s^j [S_2(k',d-j)z^{k'} - 1] \\
&= F_d(z) + z\sum_{j=1}^{d-1} s^j [F_{d-j}[z] - 1] + \frac{s^d}{1-z}
\end{aligned}$$

Therefore $\overline{N_d}(k) \leq Z_d(z)$ where $Z_d(z) = F_d(z) + z\sum_{j=1}^{d-1} s^j[F_{d-j}(z) - 1] + \frac{s^d}{1-z}$. For example, in the case of $d = 1$ and $s = 2$ we have:

$$\begin{aligned}
Z_1(z) &= F_1(z) + \frac{s}{1-z} \\
&= \frac{2z+1}{(z-1)^2} + \frac{2}{1-z} \\
&= \frac{3}{(z-1)^2}
\end{aligned}$$

Again we are able to generate the functions $Z_d(z)$ for any $d > 1$.

Using the recurrences of chapter 2, we obtain data for $S_2(k,d)$ and hence can obtain data for $\overline{N_d}(k) \leq S_2(k,d) + \sum_{j=1}^{d} s^j S_2(k-1,d-j)$. By plotting this data and the bound $F(k,d)$ on a graph, we can compare them and see that in fact $\overline{N_d}(k) \leq F(k,d)$. From figures 3.8, 3.9, 3.10 and 3.11 we can see the bound is valid when $s = 2$.

**Conjecture 1.** $\overline{N_d}(k) \leq F(k,d)$.

From the Maple implementation used to compute the asymptotic bounds, we observe that the coefficients of $F_d(z)$ and $Z_d(z)$ share the same asymptotic bound $\frac{(2s-1)^d k^d}{d!}$.

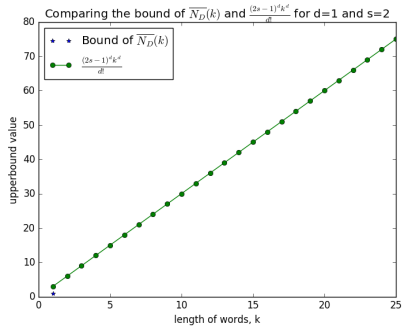


Figure 3.8: Comparison between $F(k,d)$ and the bound of $\overline{N_d}(k)$ for $d=1$ and $s=2$.

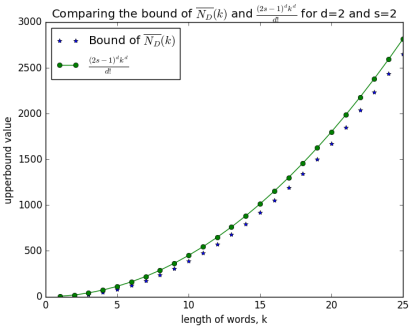


Figure 3.9: Comparison between $F(k,d)$ and the bound of $\overline{N_d}(k)$ for $d=2$ and $s=2$.

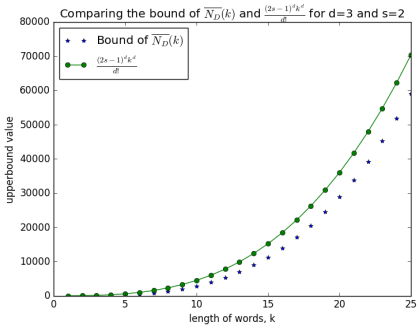


Figure 3.10: Comparison between $F(k,d)$ and the bound of $\overline{N_d}(k)$ for $d=3$ and $s=2$.

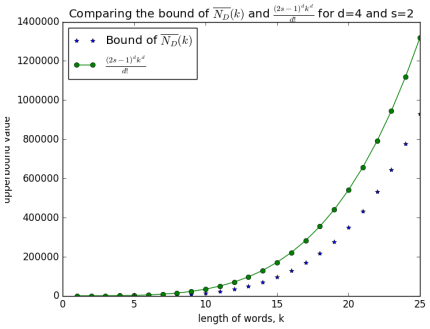


Figure 3.11: Comparison between $F(k,d)$ and the bound of $\overline{N_d}(k)$ for $d=4$ and $s=2$.

## 3.4 Comparison between $S_1(k,d)$, $S_2(k,d)$ and their asymptotic bounds on different alphabet sizes

Now that we have results for asymptotics on both $S_1(k,d)$ and $S_2(k,d)$, we want to compare those new upper-bounds on different alphabet size. Up to this point we always considered an alphabet of size 2 because it is the first non-trivial case, but for applications it would be useful to have results for $s=4$ and $s=20$. An alphabet of size four corresponds to DNA and an alphabet of size twenty corresponds to amino acids.

We start by comparing $S_1(k,d)$, $S_2(k,d)$ and $F(k,d)$. Knowing that $S_2(k,d)$ allows less redundancy than $S_1(k,d)$, we want to see how $F(k,d)$ compares to both functions. On figures 3.12, 3.13, 3.14 and 3.15 we can see that the asymptotic falls in between $S_1(k,d)$ and $S_2(k,d)$. This means adding more restrictions on the combinations of operations on the alignment was necessary to obtain a tighter bound.

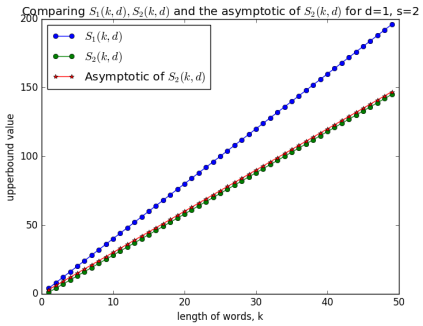


Figure 3.12: Comparison between $S_1(k,d), S_2(k,d)$ and $F(k,d)$ for $d = 1$ and $s = 2$.

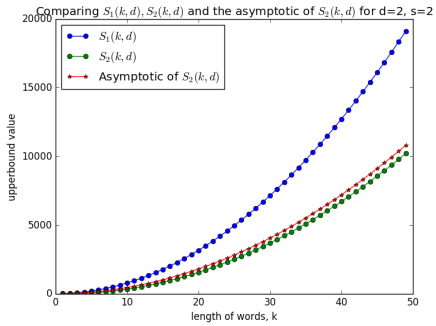


Figure 3.13: Comparison between $S_1(k,d), S_2(k,d)$ and $F(k,d)$ for $d = 2$.

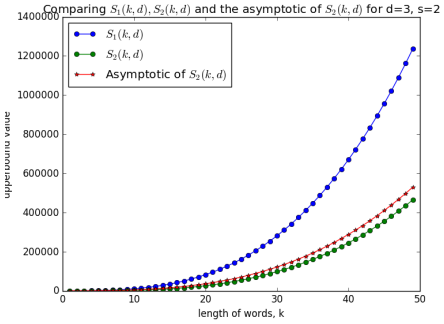


Figure 3.14: Comparison between $S_1(k,d), S_2(k,d)$ and $F(k,d)$ for $d = 3$.

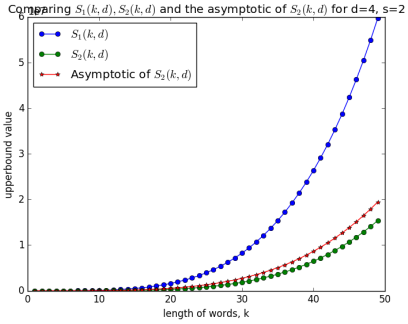


Figure 3.15: Comparison between $S_1(k,d), S_2(k,d)$ and $F(k,d)$ for $d = 4$.

To find the asymptotic of $S_1(k,d)$, we used the same method as in 3.2. We concluded that the terms of the series grow according to $\frac{(2s)^d k^d}{d!}$. On figures 3.16, 3.17, 3.18 and 3.19 we can see how the two asymptotic functions compare to one another. We see that the improved version of the recurrences ended up not having a significant impact on the asymptotic bounds. We can see the comparison between the two asymptotics on a binary alphabet on figures 3.16, 3.17, 3.18 and 3.19. As the alphabet gets larger we can see the two bounds getting closer together, as expected.

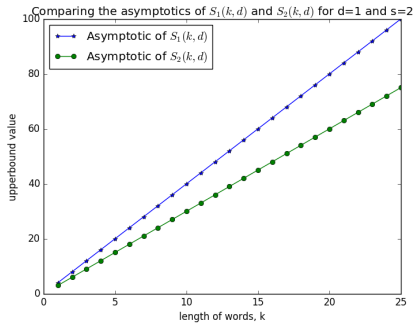


Figure 3.16: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=1, s=2$.

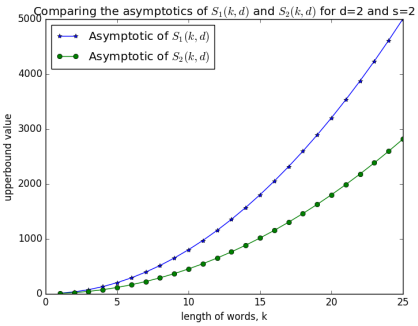


Figure 3.17: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=2, s=2$.

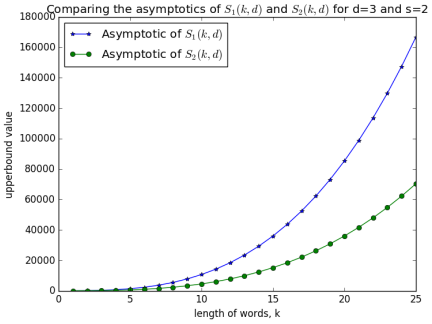


Figure 3.18: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=3, s=2$.

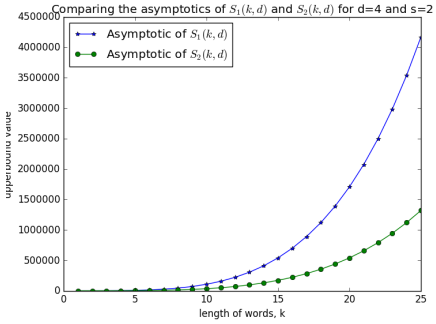


Figure 3.19: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=4, s=2$.

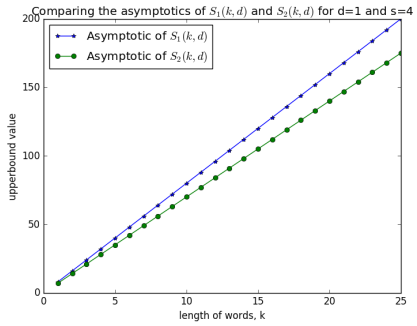


Figure 3.20: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=1, s=4$.

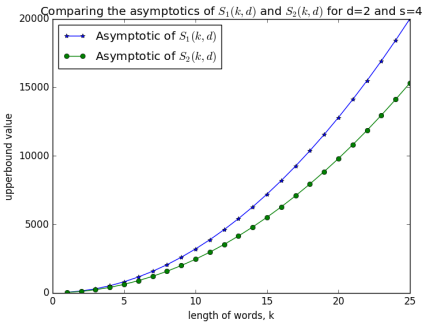


Figure 3.21: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=2, s=4$.

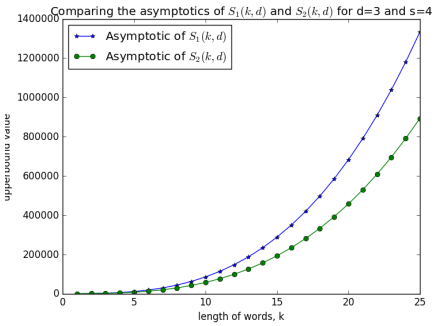


Figure 3.22: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=3, s=4$.

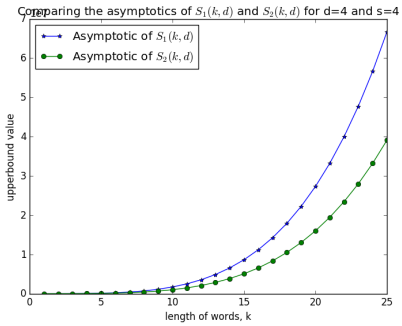


Figure 3.23: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=4, s=4$.

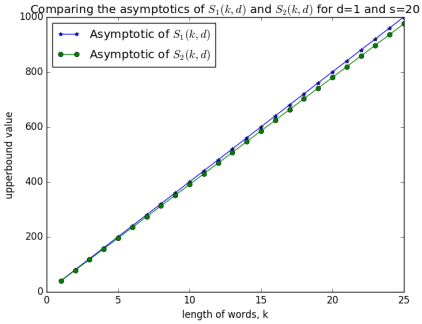


Figure 3.24: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=1, s=20$.

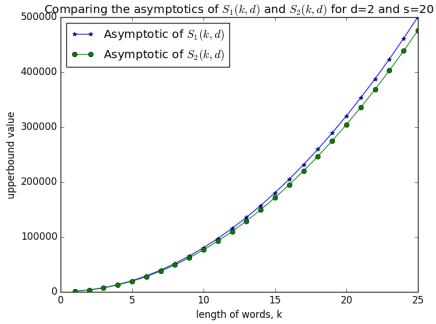


Figure 3.25: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=2, s=20$.

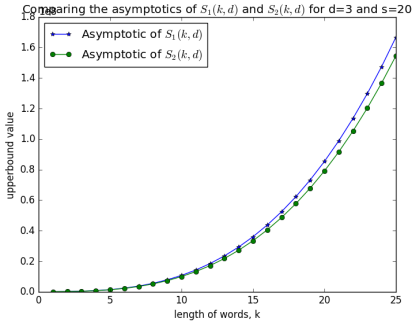


Figure 3.26: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=3, s=20$.

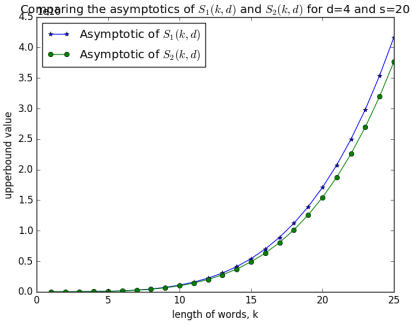


Figure 3.27: Comparison between the asymptotics of $S_1(k,d)$ and $S_2(k,d)$ on $d=4, s=20$.

## 3.5 Conclusion

From the original recurrences, we were able to determine that the bound $B(k,d,c)$ was not optimal and could be improved. Using generating functions and asymptotic analysis on the

recurrences, we were able to obtain a new bound which is polynomial in $d$. We proved that the new function $F(k,d) = \frac{(2s-1)^d k^d}{d!}$ is a much tighter bound on $S_2(k,d)$ than the original bound $B(k,d,c)$.

# Chapter 4

# On "The Levenshtein Automaton and the Size of the Neighbourhood of a Word", by Hélène Touzet

.

This chapter is dedicated to the paper [3] by Hélène Touzet, published in 2016. Instead of using the edit-script model, Touzet used finite automaton to solve exactly the neighbourhood problem. Her method allows to compute exactly the number of words in the $d$-neighbourhood of a given word $W$. Although it technically only returns the number of words from the condensed-neighbourhood, the method could be adapted to generate those words, by translating back the bit vectors into letters of the alphabet. In this chapter, we focus on explaining her method. Because this is slightly different than the outcome obtain from the edit-script model as presented by Gene Myers, we will not be able to directly compare both methods. Chapter 5 will give an idea of the work that was done in implementing both methods and the results that were obtained from both.

## 4.1 Introduction and notation

We start by defining some useful notation. In order to stay consistent with [3], the number of errors will be denoted by $k$ and the length of the initial word will be $m$. Let $\Sigma$ be a finite alphabet and let $P$ and $V$ be two strings over $\Sigma$. We will refer to $P$ as the reference word and $V$ as the word we want to transform into $P$. Since we are considering the **Levenshtein distance** between those two words, we will denote by $Lev(P, V)$ the smallest number of edit scripts needed to transform $V$ into $P$. An edit script can be one of the three operations: **substitution**, **insertion** or **deletion** of a symbol. We call a sequence of these operations an edit script.

| | **K** | **i** | **t** | **t** | **e** | **n** |
|---|---|---|---|---|---|---|
| **S** | 1 | 2 | 3 | 4 | 5 | 6 |
| **i** | 2 | 1 | 2 | 3 | 4 | 5 |
| **t** | 3 | 2 | 1 | 2 | 3 | 4 |
| **t** | 4 | 3 | 2 | 1 | 2 | 3 |
| **i** | 5 | 4 | 3 | 2 | 2 | 3 |
| **n** | 6 | 5 | 4 | 3 | 3 | 2 |
| **g** | 7 | 6 | 5 | 4 | 4 | 3 |

Table 4.1: Table of the Wagner-Fisher algorithm for P=Kitten and V=Sitting.

In the case where $P$ is fixed and we know the maximal distance $k$, we define

$$Lev(P, d) = \{\ V \in \Sigma \mid Lev(P, V) \leq d\}$$

which gives all the words at distance at most $k$ from $P$.
We also use $V_i$ to represent the $i^{th}$ letter of $V$ and $V[i..j]$ represents the substring of $V$ starting at the $i^{th}$ letter and ending at the $j^{th}$ letter.

## 4.2 Bit vector representation

We are given two strings $P$ and $V$. It is possible to compute the minimal edit distance between those two strings by computing the table of the Wagner-Fischer algorithm.

The number in the bottom-right corner of the table gives us the minimal number of edits needed to transform $V$ into $P$. In this table, the entry $(i, j)$ is defined by:

$$(i,j) = \min \begin{cases} (i-1,\ j) + 1 \\ (i,\ j-1) + 1 \\ (i-1,\ j-1) + 1^* \end{cases} \quad \text{, where } 1^* = 1 \text{ if } i \neq j \text{ and 0 otherwise.}$$

This can be translated as: moving *down* in the table represents an insertion, *right* represents a deletion and *diagonal right* represents either a substitution or an identity, depending if $1^* = 1$ or 0.

If we let $d$ be the value of the bottom-right entry of the table, then every edit script with at most $d$ errors will have a path that stays around the diagonal of the table. This diagonal will have length $2d + 1$.

We realize that in order to compute these entries of the table, we only need to know when $P_i = V_j$. This can now be represented as a grid with white and grey cells, where a

| | **K** | **i** | **t** | **t** | **e** | **n** |
|---|---|---|---|---|---|---|
| **S** | 1 | 2 | 3 | | | |
| **i** | 2 | 1 | 2 | 3 | | |
| **t** | 3 | 2 | 1 | 2 | 3 | |
| **t** | | 3 | 2 | 1 | 2 | 3 |
| **i** | | | 3 | 2 | 2 | 3 |
| **n** | | | | 3 | 3 | 2 |
| **g** | | | | | 4 | 3 |

Table 4.2: Now showing only the paths that stay around the diagonal.

white cell represents $P_i \neq V_j$ and a grey cell means equality. The idea is now to represent these horizontal lines of the table using bits. A 0 will represent a white cell and a 1 a grey cell. We will use **characteristic vectors** to represent each rows.

**Definition 1.** *Let $P \in \Sigma^*$ and let $s \in \Sigma$. The characteristic vector $\chi(s, P)$ is the bit vector of length $|P|$ such that the $i^{th}$ bit is 1 if $s = P_i$ and 0 otherwise.*

**Example 4.2.1.** $\Sigma = \{a, b, c, t\}$ , $P = cat$
$\chi(a, P) = 010$ ; $\chi(b, P) = 000$ ; $\chi(c, P) = 100$

We can use these characteristic vectors to represent an encoding of $V$. We will define the k-encoding of $V$ with respect to $P$.

**Definition 2.** *Let $P \in \Sigma^m$ and $V \in \Sigma^n$ such that $n \leq m + d$. Let $P' = \$^k P \$^{2d}$ and $V' = V\$^{m-n+d}$, where \$ is a new symbol not in $\Sigma$. The d-encoding of $V$ with respect to $P$ is the sequence of $m + d$ bit vectors of length $2d + 1$ such that the $j^{th}$ element is the characteristic vector $\chi(V'_j, P'[j - d..j + d])$.*

**Example 4.2.2.** *2-encoding of SALAD with respect to the word BALLAD*
*$P'$= \$\$BALLAD\$\$\$\$, $V'$= SALAD\$\$\$*
*2-encoding: 00000 00100 00110 10010 00010 00011 00111 01111*
*Each block represents one character of $V'$.*

Each bit vector now represents one horizontal line of the white and grey table, a zero representing a white cell and a 1 representing a grey cell. We can obtain a path in this grid by going down (which we can see as a substitution if the cell is white or as an identity if the cell is grey) moving right (a deletion) or moving left (an insertion) by one cell. This is how we represent an edit script between $P$ and $V$.

## 4.3 NULA(d)

Having this way of representing an edit script between two words, we construct the Nondeterministic Universal Levenshtein Automaton, NULA(d).

**Definition 3.** *Let $d$ be a positive number. The Nondeterministic Universal Levenshtein Automaton for $d$, NULA(d), is the NFA represented as follows:*

- *the input alphabet is* $\{0,1\}^{2d+1}$,
- *the set of states* $Q_d$ *is* $\{(x,y) \in \mathbb{N} \times \mathbb{Z}; 0 \leq x \leq d, -x \leq y \leq x\}$,
- *the transition function* $\Delta_d : Q_d \times \{0,1\}^{2d+1} \to P(Q_d)$ *is constituted of three types of transitions.*
  *insertion transitions:* $(x+1, y-1) \in \Delta_d((x,y), u)$ *for all states* $(x,y) \in Q_d$ *such that* $x < d$ *and all* $u \in \{0,1\}^{d+y}0\{0,1\}^{d-y}$
  *substitution transitions:* $(x+1, y) \in \Delta_d((x,y), u)$ *for all states* $(x,y) \in Q_d$ *such that* $x < d$ *and all* $u \in \{0,1\}^{d+y}0\{0,1\}^{d-y}$
  *deletion + identity transitions:* $(x+l, y-l) \in \Delta_d((x,y), u)$ *for all states* $(x,y) \in Q_d$, *all* $l$ *such that* $0 \leq l \leq d-x$ *and all* $u \in \{0,1\}^{d+y}0^l1\{0,1\}^{d-y-l}$
- *the start state is* $(0,0)$,
- *all states are accepting.*

We will define the states of the automaton based on the lanes of the previous grid construction. We need a way to keep track of the number of errors and of which operations we have done so far. The later can be represented by which lane of the grid we moved to. The states we define are of the form $(x, y)$, where $x$ is the number of errors made so far and $y$ corresponds to which lane we are standing in. We will use the bit vectors for the transitions on the automaton. At each transition, we read one bit vector, which corresponds to reading one letter of $V$ each time we move in NULA(d). In order to reduce the nondeterminism of the automaton, the possible transitions it accepts are:

- insertion of $v$
- substitution of $v$
- a sequence of $l$ deletion followed by an identity with $v$

**Definition 4.** *The right language of a state $q \in Q_d$ is the set of all sequences of bit vectors $u$ such that NULA(d), when started at $q$ will accept $u$. It is denoted by $L(q)$.*

**Corollary 1.** *Let $P$, $V \in \Sigma^*$. Lev(P,V) $\leq d$ if and only if the d-encoding of $V$ with respect to $P$ is accepted by NULA(d).*

## 4.4 DULA(d)

From NULA(d), it is now possible to construct a Deterministic Universal Levenshtein Automaton, DULA(d). In order to do so, we need to use the powerset construction that

converts a Nondeterministic Finite Automaton into a Deterministic one. The method goes as follows: we initialize with the starting state of NULA, {(0,0)}. We perform every possible transitions from (0,0) and get new set of states for each of them. We proceed for all the set of states that we obtain until we've tried every path of the automaton.

**Example 4.4.1.** *Transforming NULA(1) into DULA(1)*
*initial state: {(0,0)}*

| *starting states* | *transitions* | *ending states* |
|---|---|---|
| *{(0,0)}* | $*1*$ | *{(0,0)}* |
| | $*00$ | *{(1,-1),(1,0)}* |
| | $*01$ | *{(1,-1),(1,0),(1,-1)}* |
| *{(1,-1),(1,0)}* | $10*$ | *{(1,0)}* |
| | $11*$ | *{(1,-1),(1,0)}* |
| | $01*$ | *{(1,0)}* |
| *{(1,-1),(1,0),(1,1)}* | *100* | *{(1,-1)}* |
| | *010* | *{(1,0)}* |
| | *001* | *{(1,1)}* |
| | *110* | *{(1,-1),(1,0)}* |
| | *011* | *{(1,0),(1,1)}* |
| | *101* | *{(1,-1),(1,1)}* |
| *{(1,0)}* | $*1*$ | *{(1,0)}* |
| *{(1,-1)}* | $1**$ | *{(1,-1)}* |
| *{(1,1)}* | $**1$ | *{(1,1)}* |
| *{(1,0),(1,1)}* | $*10$ | *{(1,0)}* |
| | $*11$ | *{(1,0),(1,1)}* |
| | $*01$ | *{(1,1)}* |
| *{(1,-1),(1,1)}* | $1*0$ | *{(1,-1)}* |
| | $1*1$ | *{(1,-1),(1,1)}* |
| | $0*1$ | *{(1,1)}* |

Hence we obtain that the possible states for DULA(1) are: {(0,0)}, {(1,0)}, {(1,-1)}, {(1,1)}, {(1,-1),(1,0)}, {(1,0),(1,1)}, {(1,-1),(1,1)}, {(1,-1),(1,0),(1,1)}.
In order to simplify notation, we will relabel the states in the following way:

| set of states | new label |
|---|---|
| {(0,0)} | 0 |
| {(1,-1),(1,0),(1,1)} | 1 |
| {(1,0),(1,1)} | 2 |
| {(1,-1),(1,1)} | 3 |
| {(1,-1),(1,0)} | 4 |
| {(1,1)} | 5 |
| {(1,0)} | 6 |
| {(1,-1)} | 7 |

At this point it is possible to reduce the number of states (in each set). In order to do so, we introduce the notion of *subsumed states*.

**Definition 5.** *Let $(x, y)$ and $(x', y')$ be two states of $Q_d$. We say that $(x, y)$ subsumes $(x', y')$ if $x < x'$ and $y + x - x' \leq y' \leq y + x' - x$. We denote it by $(x', y') \sqsubset (x, y)$.*

**Example 4.4.2.** *(1,0) $\sqsubset$ (0,0) and (2,-1) $\sqsubset$ (1,0)*
*Using the subsumed states notion, we define REDUCED($Q'$) as the largest subset of $Q'$ such that no two subsets are subsumed, where $Q'$ is a subset of $Q_k$.*

We will define REDUCED($Q_d$) as the set of all $Q$'s such that $Q$=REDUCED($Q$),where each $Q$ is a subset of $Q_d$. This should also be the set of states of DULA(d).

## 4.5 The counting problem

Now that we have a deterministic automaton, the only problem left is to re-convert the transitions back into words over $\Sigma$. To do so, we will construct a new deterministic automaton, Encod$(P, d)$, that will represent the possible k-encoding of words with respect to $P$.

Recall that $\overline{Lev}(P, V)$ is the set of bit vector sequences $u$ in $(\{0,1\}^{2d+1})^*$ such that there exist $V$ in $Lev(P, d)$ whose d-encoding with respect to $P$ is $u$. Note that $\overline{Lev}(P, V) =$ DULA$(d) \cap$ Encod$(P, d)$. It will be possible with $\overline{Lev}(P, V)$ to compute the neighbourhood of the word $P$.

**Definition 6.** *Let P be a string of $\Sigma^*$. Encod(P,d) is the set*
*$\{u \in (\{0,1\}^{2d+1})^* |\ \exists V \in \Sigma^*$ s.t. u is the d-encoding of V with respect to P $\}$*

Also note that the elements of Encod(P,k) have length $2d + 1$.
To define the automaton that accepts the language Encod(P,d), we need one more definition.

**Definition 7.** *Let $V \in (\Sigma \cup \{\$\})^*$. Define $B(V) = \{\chi(s, V); s \in \Sigma\}$*

**Example 4.5.1.** *Let* $\Sigma = \{A, B, L\}$. *B(AAB)={000,110,001}, B(ABL)={100,010,001}.*

$B(V)$ has the following properties:

1. If $V_i = V_j$ then $u_i = u_j$, in each bit vector $u$.
2. If $u_i = 1$ and $u_j = 1$, then $V_i = V_j$.
3. If $V_i = \$$, then $u_i = 0$.

The automaton for Encod$(P, d)$ will have $(\{0,1\}^{2d+1})^*$ as input alphabet and its states will be
$\{0, .., m+d\} \cup \{\$_{m-d+1}, .., \$_{m+d}\}$ with 0 being the starting state. The two accepting states will be $m+k$ and $\$_{m+d}$. The transitions will be defined by the function $\gamma$, where:

$$\gamma(i-1, u) = i, \text{where } 1 \leq i \leq m+d \text{ and } u \in B(P'[i-d..i+d])$$
$$\gamma(i-1, 0^{d+1+m-i}1^{i+d-m}) = \$_i, \text{where } m-d+1 \leq i \leq m+d$$
$$\gamma(\$_{i-1}, 0^{d+1+m-i}1^{i+d-m}) = \$_i, \text{where } m-d+1 \leq i \leq m+d$$

In other words, the states in $\{0, ..., m+d\}$ recognizes the encoding of words that do not contain \$ signs. When there is we encounter the first \$ we move to the state $\$_?$.

The final problem we need to fix with $\overline{Lev}(P, V)$ is the reconversion into the initial alphabet. Because we have used bit vectors so far, we cannot only count the number of bit vectors per transition as it is possible that multiple letters are represent by the same bit vector. Let us define a function $\alpha : \{0,1\}^{2d+1} \times \Sigma^{2d+1} \to \mathbb{N}$ that will count the number of symbols $s \in \Sigma$ such that $\chi(s, V) = u$. The input of $\alpha$ represents the bit vector $u$ of length $2d+1$ and the substring of $V$, also of length $2d+1$. We define $\alpha$ as follows:

$$\alpha(u, V) = 1, \text{when at least one bit of u is 1}$$
$$\alpha(u, V) = |\Sigma| - |V|_\Sigma, \text{otherwise}$$

Recall that $|V|_\Sigma$ is the number of distinct symbols of $\Sigma$ in $V$. Hence when $u = 00\cdots00$ we count the number of possible symbols of $\Sigma$ that can generate that specific string.

## 4.6 Recurrences

We will use recurrence relations to count the number of words in the neighbourhood of a given word $P$. The automaton $\overline{Lev}(P, d) = DULA(d) \cap Encod(P, d)$ is deterministic, so

| | i=0 | i=1 | i=2 | i=3 |
|---|---|---|---|---|
| S(0,0)=1 | S(0,1)=1<br>S(4,1)=2 | S(0,2)=1<br>S(4,2)=4 | S(0,3)=1<br>S(4,3)=6 | <br>S(4,4)=3<br>S(7,4)=6 |
| | | | S(1,$\$_3$)=1 | S(0,$\$_4$)=1<br>S(2,$\$_4$)=1<br>S(6,$\$_4$)=6 |

Table 4.3: Recurrence relation of $\overline{Lev}(AAA, 1) = DULA(1) \cap Encod(AAA, 1)$

counting the number of paths leading to a state will give us the number of words accepted by this state. Therefore we will count the number of words accepted by the final states of this automaton. It can be done by defining a function $S$ as follows:

$$S : REDUCED(Q_d) \times (\{0, ..., m+d\} \cup \{\$_{m-d+1}, ..., \$_{m+d}\}) \to \mathbb{N}$$

$$\begin{aligned}
S(0,0) &= 1 \\
S(q', i+1) &= \sum_{u,q;q'=\delta_d(q,u)} \alpha(u, P'[i-d..i+d]) \times S(q,i), \text{ for } 0 \leq i < m+d \\
S(q', \$_{m-d+1}) &= \sum_{u,q;q'=\delta_d(q,u)} S(q, m-d) \\
S(q', \$_{i+1}) &= \sum_{u,q;q'=\delta_d(q,u)} S(q,i) + S(q, \$_i), \text{ for } m-d \leq i < m+d
\end{aligned}$$

The function take as entry a state of DULA(d) and a state of Encod(P,d), and will return an integer that represent the number of path that leads to the input state of DULA(d) by using the transition to get to the input state of Encod(P,d). In other words, S(1,1) would count the number of paths leading to state 1 of DULA(d) from state 0 using the first transitions of Encod(P,d).

The idea is that for each state of the form $(q', i+1)$,we count the number of letters associate to each transition $i$ of Encod$(P, d)$ to get to the state $q'$ in DULA$(d)$ from the state $q$, times the number of paths leading to $q$. States of the form $(q', \$_{m-d+1})$ represent the first possible encounter of a \$ in the word $V$ and is the same as the previous state $(q, m-d)$. Finally, states of the form $(q', \$_{i+1})$ are the sum of the previous \$ state as well as the previous alphabet state $(q, i)$ since these are the two possible way we can move to the $\$_{i+1}$ state.

The final result gives us a way to count the number of words in the neighbourhood of $P$.

**Proposition 1.** $|Lev(P,d)| = \sum_{q \in REDUCED(Q_d)} S(q, m+d) + S(q, \$_{m+d})$

## 4.7 Conclusion

This method allows to compute the exact number of words in the $d$-neighbourhood of a word $P$. Starting with a Nondeterministic Automaton, we construct a Deterministic version which in turn gives us a product automaton. Although it solves exactly the initial problem, the exponential computation time makes this approach impractical. It is conjectured in [3] that the number of states in DULA(d) is in $O(7^d)$. Because of this exponential growth of the states, and DULA(d) being necessary to the computations of the final automaton $\overline{Lev}(P,d)$, it makes the computation of data difficult for large values of $d$.

# Chapter 5

# Comparison between Touzet and Myers methods

This chapter will focus on the comparison between the results obtained from the methods presented in chapters 2 and 4. From the finite deterministic automaton method of Touzet, we obtain the exact size of the $d$-neighbourhood of a word, while the alignment method of Myers gives an upper-bound on the size of the condensed-neighbourhood.

Because these two methods apply to slightly different concepts, we will transform the recurrences of Myers to generate edit-scripts. This allows us to remove the edit-scripts that are generated multiple times. Once we've removed the redundant scripts, we get a bound on the condensed-neighbourhood which counts only distinct edit-scripts. We will compare this bound with the exact results on the size of the $d$-neighbourhood.

## 5.1 Results of experiments

To compute both methods, we implemented Myers recurrences and Touzet's automaton using Python. To verify that the results obtain by those implementation were correct, we used Maple to double-check some functions.

With the alignment method, we were able to generate the upper-bound given by the recurrences for given values of $d$ and $k$ as well as a specified length of the alphabet. For the purpose of the experiment, we decided to keep a binary alphabet, but this could easily be adapted to larger cases. Starting with the initial recurrences that give the bound on $\overline{N_d}(k)$ , we were able to modify them to generate edit-scripts. From there, we were able to identify which ones were repeated as well as which one were part of the condensed-neighbourhood.

On the other hand, the implementation of the automaton method gave us the actual number of words in the $d$-neighbourhood for all the words on of length $k$ a binary alphabet. These results allowed for a comparison between the exact value and the number generated by the bound of Myers.

On figures 5.1, 5.3 and 5.5 we show the results obtain for he exact value of the neighbourhood. All words of length $k$ were generated as well as the size of their respective neighbourhoods and we used box plots to represent the values. We can see the average size of the neighbourhood for the words of length $k$ in red. On figures 5.2, 5.4 and 5.6 we see the number of distinct words generated by the recurrences of $\overline{N_d}(k)$ . Again we have the average for all words of length $k$.

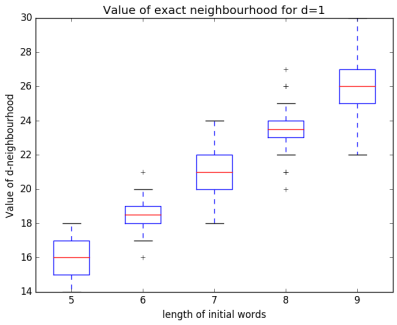


Figure 5.1: Value of the exact 1-neighbourhood of words of length 5 to 9 on an alphabet of size 2.

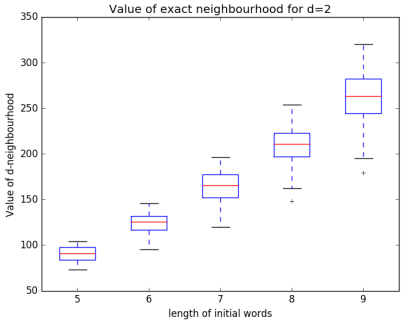


Figure 5.3: Value of the exact 2-neighbourhood of words of length 5 to 9 on an alphabet of size 2.

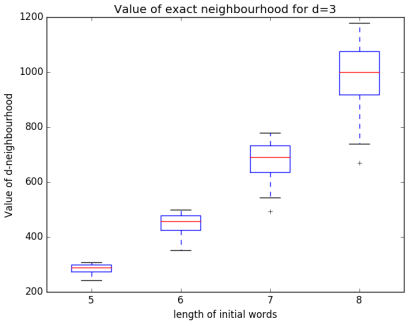


Figure 5.5: Value of the exact 3-neighbourhood of words of length 5 to 9 on an alphabet of size 2.

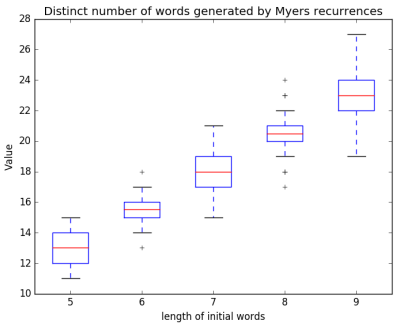


Figure 5.2: Number of distinct words generated by the recurrences $\overline{N_d}(k)$ for words of length 5 to 9 on an alphabet of size 2 with d=1.

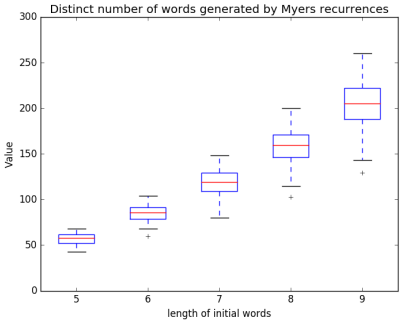


Figure 5.4: Number of distinct words generated by the recurrences $\overline{N_d}(k)$ for words of length 5 to 9 on an alphabet of size 2 with d=2.

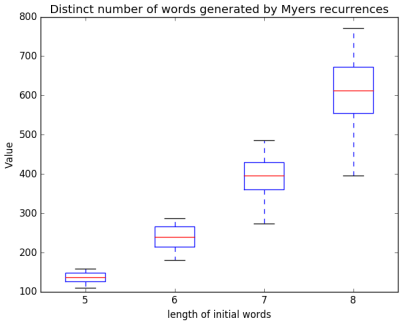


Figure 5.6: Number of distinct words generated by the recurrences $\overline{N_d}(k)$ for words of length 5 to 9 on an alphabet of size 2 with d=3.

If we look at the two figures for the case $d = 1$, we see clearly the results generated by $\overline{N_d}(k)$

# Chapter 6

# Conclusion

The problem of computing the size of the condensed-neighbourhood of a word of length $k$ was well studied by Gene Myers in [1, 2]. We studied both versions of his recurrences to give an upper-bound on the size of the condensed-neighbourhood and were able to derive asymptotic bounds for both. After studying the behavior of these bounds on alphabets of size 4 and 20 we determined that $F(k,d) = \frac{(2s-1)^d k^d}{d!}$ was appropriate to bound the recurrences $S_2(k,d)$. In the future we would like to study the impact of those recurrences on the probability of identifying a word of the condensed-neighbourhhod in an indexed text. Myers presented in [1] bounds that rely on his function $B(k,d,c)$ for this probability, and now that we have found a tighter bound on the recurrences it would be good see how it compares for the probabilities. As these probabilities have an impact on the complexity of the algorithm, this would be relevant to study.

# Bibliography

[1] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4-5):345–374, 1994.

[2] Gene Myers. What's Behind Blast. In Cédric Chauve, Nadia El-Mabrouk, and Eric Tannier, editors, *Models and Algorithms for Genome Evolution*, pages 3–15. Springer, 2013.

[3] Hélène Touzet. On the Levenshtein automata and the size of the neighbourhood of a word. In *Language and automata theory and applications*, volume 9618 of *Lecture Notes in Comput. Sci.*, pages 207–218. Springer, [Cham], 2016.

# Appendix A

# Code

Link to GitHub:

https://github.com/Francepnadeau/Final_Project_2017