FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Multi-threaded implementation of "Four Russians" edit distance algorithm
Student:	Martin Rejmon
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2019/20

Instructions

1) Become familiar with different algorithms for computing edit distance and similar problems [1].
2) Examine the Four Russians' edit distance algorithm in great detail [2,3,4].
3) Discuss possibilities for multithreaded parallelization of this algorithm.
4) Implement the Four Russians' edit distance algorithm in C or C++ using OpenMP API.
5) Evaluate the implementation experimentally and compare it with existing implementations of other algorithms for computing edit distance.

References

[1] Aho, Hopcroft, Ullman: The Design and Analysis of Computer Algorithms 1st Edition, 978-0201000290, Amazon Books
[2] A. H. Mikkelsen: Local alignment using the Four-Russians technique, 20106624, Master's Thesis, Computer Science, Aarhus university, June 2015
[3] Y. Frid, D. Gusfield:An improved Four-Russians method and sparsified Four-Russians algorithm for RNA folding, Algorithms Mol Biol. 2016;11:22
[4] Youngho Kim, Joong Chae Na,Heejin Park,Jeong Seop Sim: A space-efficient alphabet-independent Four-Russians' lookup table and a multithreaded Four-Russians' edit distance algorithm, Theoretical Computer Science Volume 656, Part B, 20 December 2016

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

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 11, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Multi-threaded implementation of the Four-Russians edit distance algorithm

Martin Rejmon

Department of Theoretical Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

May 14, 2019

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 14, 2019 ……

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Rejmon, Martin. *Multi-threaded implementation of the Four-Russians edit distance algorithm.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Editační vzdálenost lze vypočítat s obecně známým algoritmem využívajícím dynamické programování v čase $\mathcal{O}(n^2)$, kde n je délka vstupních řetězců. Algoritmus Čtyř Rusů zlepšuje tuto složitost s pomocí vyhledávací tabulky o faktor $\log^2 n$. V této práci je tento algoritmus podrobně prozkoumán a důležité implementační detaily jsou prodiskutovány, přičemž zvláštní ohled je brán na paralelizování algoritmu a zmenšení velikosti vyhledávací tabulky. Implementace v jazyce C++ je poskytnuta a její výkon je porovnán v několika experimentech s populární knihovnou na výpočet editační vzdálenosti. Výsledky naznačují, že algoritmus je v praxi použitelnou volbou, ale není optimální.

Klíčová slova editační vzdálenost, algoritmus Čtyř Rusů, implementace, paralelizace

Abstract

Edit distance can be computed with the well-known dynamic programming algorithm in $\mathcal{O}(n^2)$ time, where n is the length of the input strings. The Four-Russians algorithm improves this complexity by a factor of $\log^2 n$ by using a lookup table. In this thesis, the algorithm is thoroughly examined and important implementation details are discussed, with special consideration given to parallelizing the algorithm and reducing the size of the lookup table. An implementation in C++ is provided and its performance is compared with a popular edit distance library in several experiments. The results indicate that the algorithm is a viable option in practice, but not optimal.

Keywords edit distance, Four-Russians algorithm, implementation, parallelization

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Edit distance

The edit distance between two strings is the smallest number of edit operations needed to transform one string into the other. The three possible edit operations are: insertion of any character into the first string, deletion of any character from the first string and substitution of any character from the second string for any character from the first string. For example, the edit distance between the two words *survey* and *surgery* is 2 and the corresponding edit operations are the substitution of g for v and the insertion of r before the last character. In other words, edit distance is a quantification of the dissimilarity of two strings. Such measure has applications in numerous problems in computer science, for example in bioinformatics, spell checking, speech recognition, search engines, databases, and natural language processing, thus finding ways to compute it in the fastest way possible is highly desirable.

Before we move on further, it has to be said that this definition for edit distance is sometimes called the Levenshtein distance, with edit distance referring to a larger family of string distances. That being said, the name edit distance is used in this thesis to stay consistent with the cited sources. Furthermore, any other variations of edit distance, such as the weighted edit distance, where edit operations have varying costs depending on the characters upon which they are operating, or the Hamming distance, where only replacements are allowed, or even the Damerau–Levenshtein distance, where transposition is also possible (highly relevant for the problem of typing errors), are not discussed in this thesis. Similarly, no attention is paid to actually retrieving the sequence of edit operations leading to the resulting edit distance. While the focus of the thesis is on computing edit distance, the described algorithm can easily be converted to solve the problem of approximate string matching and with a little more difficulty used for the purposes of local sequence alignment.

Four-Russians

The Four-Russians algorithm for computing edit distance is actually an application of the Four-Russians technique on the dynamic programming algorithm for computing edit distance (which will itself be introduced in the first chapter). The Four-Russians technique can be used to speed up algorithms operating on matrices with a limited amount of possible values in each cell by storing computed submatrices in a lookup table. The name of this peculiarly named technique refers to its inventors, as it was presented in 1970 by V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzhev [1], who used it to construct the transitive closure of a directed graph. Of course, what we are more interested in is its application on the problem of edit distance, which was done in 1980 by Masek and Paterson [2]. This lead to the aforementioned time complexity of $\mathcal{O}((n^2)/\log^2 n)$. To date, this algorithm has the best worst-case time complexity for computing edit distance, which is emphasized further by a fact that was proven by Backurs and Indyk in 2015 [3]: the edit distance between two strings cannot be computed in strongly subquadratic time (unless SETH (strong exponential time hypothesis) is false), in other words, the edit distance between two strings of size n cannot be computed in $\mathcal{O}(n^{2-\epsilon})$ time for any $\epsilon > 0$.

However, two conditions have to be met for this time complexity to hold. The first one is that the costs of edit operations have to be a multiple of a single positive real number, which we fulfill by focusing solely on unit cost edit distance, and the second one is that the alphabet from which the strings are has to be finite. This condition is avoided when a clever string encoding described in 2016 by Kim et al. [4] is used, which also significantly reduces the size of the lookup table. Actually, an improved version of their string encoding is introduced in this thesis, which lowers the size of the lookup table even more. Unfortunately, both the original and the improved string encoding make the algorithm lose one logarithm factor of speedup. While that is undesirable, it is needed to make the algorithm usable in practice for alphabets of size > 4. To offset this issue, Kim et al. presented a few ideas about how to parallelize the algorithm, which we will also discuss later.

One of the major parts of this thesis is the actual implementation of the Four-Russians algorithm. This implementation was mainly used to evaluate the practicality of the algorithm. When the algorithm was first conceived it was purely theoretical, as computers at the time did not actually have enough memory to contain the required lookup table. The situation is different nowadays, nonetheless, the question this thesis is trying to answer still stands: is the Four-Russians algorithm in practice a competitive option for computing edit distance?

Structure of the thesis

The thesis is split into four chapters. In Chapter 1 several algorithms for computing edit distance are summarized and the relationship between computing edit distance and approximate string matching is mentioned. Chapter 2 explains the main idea that makes the Four-Russians algorithm viable in practice and improves the viability further by resolving the dependence of the size of the lookup table on the size of the input strings' alphabet. On top of that, the possibilities for parallelization are presented. The obtained speedup is observed in Chapter 3, together with a description of how to use modulo arithmetic to cut down on the size of the lookup table. In addition to that, some consideration is given to the recommended values for the parameters required by the algorithm. Chapter 4 concludes the thesis by empirically evaluating the Four-Russians algorithm. Viability is ascertained by comparing it to the dynamic programming algorithm and possible optimality by comparing it to a state-of-the-art algorithm.

Chapter 1

Overview of Options

1.1 Basic definitions

Let arrays, vectors, and matrices be indexed from zero and strings indexed from one. If S is a string, let $S[i \dots j]$ be equal to the characters $(i, i+1, \dots, j)$, or to the empty string if $j > i$. Let U and V be strings of sizes m and n over some alphabet Σ.

1.2 Wagner and Fischer, 1974

The dynamic programming algorithm for computing edit distance was presented by Wagner and Fischer in 1974 [5]. Let D be a matrix of size $(m+1) \times (n+1)$, where $D[i, j]$ represents the edit distance from $U[1 \dots i]$ to $V[1 \dots j]$. The pseudocode is presented in Algorithm 1. Function Comparison$(a,\ b)$ returns 0 if a and b are equal and 1 otherwise. Example of a filled out D can be seen in Figure 1.1.

Algorithm 1 Wagner-Fischer

```
1:  for i ← 0 to n do                                   ▷ Initialize first row
2:      D[0, i] ← i
3:  for j ← 1 to m do                                   ▷ Initialize first column
4:      D[j, 0] ← j
5:  for i ← 1 to n do                                   ▷ Compute the matrix
6:      for j ← 1 to m do
7:          tmp1 ← D[i − 1, j] + 1
8:          tmp2 ← D[i, j − 1] + 1
9:          tmp3 ← D[i − 1, j − 1] + Comparison(U[i], V[j])
10:         D[i, j] ← min(tmp1, tmp2, tmp3)
11: return D[m, n]
```

		B	B	C	A	B	C
	0	1	2	3	4	5	6
A	1	1	2	3	3	4	5
B	2	1	1	2	3	3	4
B	3	2	1	2	3	3	4
B	4	3	2	2	3	3	4
A	5	4	3	3	2	3	4
C	6	5	4	3	3	3	3

Figure 1.1: Matrix used for computing edit distance
lnput strings are $U = ABBBAC$ and $V = BBCABC$. The highlighted cells trace the sequence of edit operations. Going down represents deletion, going right insertion, and going bottom right represents substitution (where same character substitution is a zero cost operation).

The paraphrase of the proof of the correctness of the algorithm is omitted for brevity. Let us at least intuitively state, that when we are looking for the shortest edit sequence, it makes sense for each character to be associated with maximally one edit operation, and as such the resulting edit sequence can be ordered from left to right. Then let us imagine that we are trying to compute the cell $D[m,\ n]$. We have three options and we choose the one which gives the best result:

- the character $U[m]$ is deleted from the end, we get $D[m-1,\ n]+1$
- the character $V[n]$ is inserted at the end, we get $D[m,\ n-1]+1$
- character $V[n]$ is substituted for $U[m]$, $D[m-1,\ n-1]+(U[m] \neq V[n])$

On the other hand, it is easy to see that the algorithm has time and memory complexity of $\mathcal{O}(mn)$. It's similarly easy to see that it can be modified so that only the last two rows/columns (or even only the last one) are kept in memory. This reduces the memory complexity to $\mathcal{O}(\min(m,n))$.

This algorithm will be referred to from now on as the naive algorithm.

1.3 Masek and Paterson, 1980

As this algorithm by Masek and Paterson [2] is the focal point of this thesis, it is analyzed in more detail, including time complexity and such, in the next chapter. However, a brief overview is also presented here for completeness.

The main idea is that the matrix from the naive algorithm is split into many small overlapping blocks, as can be seen in Figure 1.2. The first row, the first column, and the accompanying string substrings are the input values of a block and all the remaining values are the output values, which are

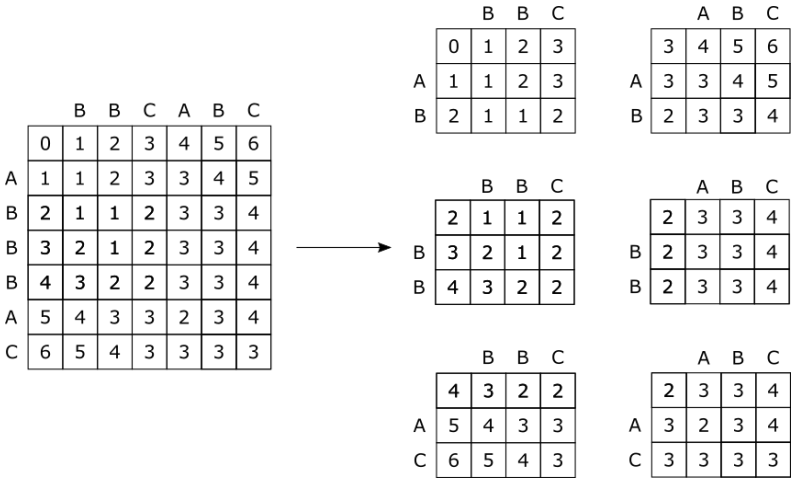

Figure 1.2: Matrix split into overlapping blocks

determined solely by the input values. In the first step of the method (called the preprocessing step) all possible combinations for the input values are generated and for each one the corresponding output values are computed and stored in a lookup table (from now on LUT) (to be accurate, we only save the last column and the last row of a block).

In the second step (called the computation step) we proceed similarly as in the naive algorithm. We initialize a matrix of blocks and set the first row and the first column to trivially known values. Afterward, we iterate over each block, but instead of computing the values we use the input values to look up the output values in the LUT. Because the blocks overlap, we can use these values as the input values for the subsequent blocks. If the blocks do not fit perfectly in the matrix, we can use the naive algorithm to compute the remaining rows and/or columns.

1.4 Ukkonen, 1985

In 1985, Ukkonen presented two algorithms for computing edit distance in $\mathcal{O}(s \min(m, n))$ time and $\mathcal{O}(\min(s, m, n))$ space, where s is the edit distance between n and m [6]. We are going to describe the first one and omit the second one, as although it is faster in practice, it as also quite nontrivial.

First, let's imagine that we only care what the precise value of the edit distance is if it is smaller than k, for some parameter k and that $m = n$. In the naive algorithm, the final cell $D[m,\ n]$ will be on the same diagonal as the starting cell, that is $D[0,\ 0]$. While computing edit distance we can only move between diagonals with insertion or deletion, both of which have to increase the resulting edit distance by one. As we have to end up on the same diagonal we started, the resulting sequence of edit operations will be located only on a

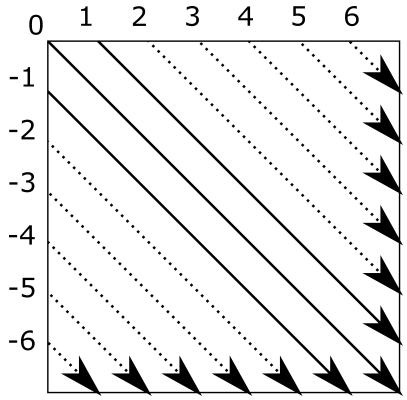

Figure 1.3: Band of diagonals computed by the Ukkonen's algorithm
Example for $m = n = 6,\ k = 3$. Dotted diagonals will be skipped. Numbers represent the indices of the diagonals.

certain band of diagonals in the middle of the matrix, limited by the value k, and we can skip the computation of the rest of the diagonals, that is, we act as if their values are equal to ∞. More precisely, if we index diagonals such that their index is equal to the value of $j - i$ of all their cells, where i and j are of course the indices of the cells, we only have to compute diagonals from $\lfloor -k/2 \rfloor$ to $\lfloor k/2 \rfloor$. This observation is illustrated in Figure 1.3. Furthermore, we can deduce whether the value in $D[m,\ n]$ is the correct result by simply checking if it is smaller than k.

As there are only $2\lfloor k/2 \rfloor + 1$ of these diagonals and each one has $\mathcal{O}(\min(m, n))$ entries, we get a time complexity of $\mathcal{O}(k \min(m, n))$ and as we only have to keep in memory two rows of diagonals, we get a space complexity of $\mathcal{O}(\min(k, m, n))$. Now, the math gets a little more convoluted if $m \neq n$. If $m > n$, we compute diagonals from $n-m-h$ to h, where $h = \lfloor (k-|n-m|)/2 \rceil$, and if $m \leq n$, we compute diagonals from $-h$ to $n - m + h$, where h is the same as before. That being said, the complexities remain the same.

Now, we just call this algorithm with values of k as following: $k_0 = |n - m| + 1,\ k_1 = 2k_0,\ \ldots, k_r = 2^r k_0$, until we get a correct result. The time complexity of this solution is $\mathcal{O}((\sum_0^r k_i) \min(m, n))$, that is $\mathcal{O}(k_r \min(m, n))$, and since $s > k^r/2$, it is also $\mathcal{O}(s \min(m, n))$.

1.5 Approximate string matching

The goal of approximate string matching (also colloquially known as fuzzy string searching) is, given a pattern of size m and a text of size n, to find all positions in the text where the pattern matches with a limited amount of errors k. The problem of approximate string matching is very closely linked with the problem of computing edit distance. This means that some algorithms for

approximate string matching can be very easily converted to computing edit distance and vice versa. For example, in the naive algorithm we only need to initialize the first row to zeroes, as the pattern can start anywhere in the text, and then all the ending positions of all the matches are going to be equal to the column indices of the cells in the last row that have smaller values than k – the allowed number of errors. The beginning positions can be recovered by reconstructing the sequences of edit operations.

Another algorithm that can be easily repurposed is the bit-vector algorithm introduced by Myers in 1998 [7], which is considered one of, if not the fastest algorithm in practice. For these reasons, it will be used in the last chapter of this thesis for evaluating the usefulness of the Four-Russians algorithm. It works by converting a part of a column of the matrix of the naive algorithm into several vectors of bits and then using clever combinations of bitwise operations to calculate the following part of the next column. As such it computes up to w elements at a time, where w is the size of a computer word, which means the time complexity is $\mathcal{O}(mn/w)$. Furthermore, the search space is reduced by a similar technique as in the last algorithm by Ukkonen (and this technique was coincidentally also invented by Ukkonen), and the time complexity is then $\mathcal{O}(kn/w)$.

Chapter 2

Four-Russians Fully Reviewed

2.1 Parts of a block

Let each block be of size $(t_m + 1) \times (t_n + 1)$ for some parameters t_m and t_n, where t_m and t_n are positive integers. Let K be the top left corner of a block, A the rest of the first column ($|A| = t_m$), B the rest of the first row ($|B| = t_n$), A' the last column without the first element ($|A'| = t_m$), B' the last row without the first element ($|B'| = t_n$), X the substring of U ($|X| = t_m$) and Y the substring of V ($|Y| = t_n$). An example can be seen in the first two images in Figure 2.1.

The values in each element of A, B and K range from 0 to $\max(m, n)$ and the elements in X and Y can contain any characters from Σ. This leads us to a number of combinations equal to $(\max(m, n))^{t_m+t_n+1}|\Sigma|^{t_m+t_n}$, which is already much more than for example the number of cells that the naive algorithm has to compute. To make the algorithm more practical we have to limit the number of possible values in A, B and K. Masek and Peterson have proven [2] that each cell in the matrix D can differ from each neighboring cell by at most 1.

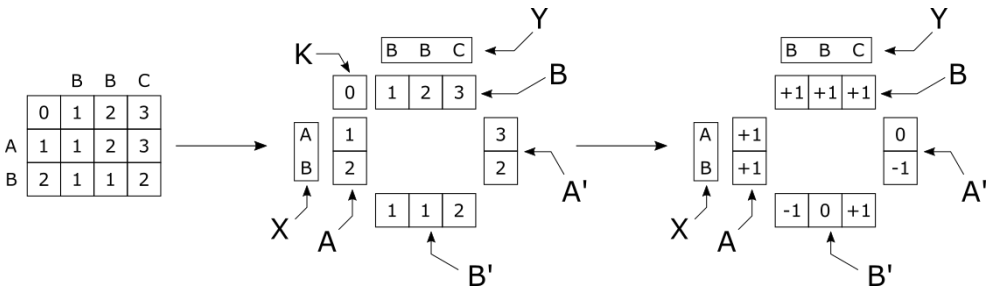

Figure 2.1: Detailed description of a block

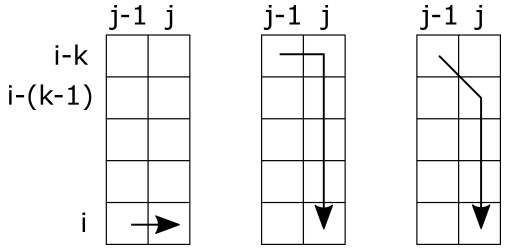

Figure 2.2: Cases for the proof of differences between the elements of the matrix

We can rewrite that as the following equation

$$D[i,\ j-1]-1 \leq D[i,\ j] \leq D[i,\ j-1]+1.$$

The upper bound, $D[i,\ j] \leq D[i,\ j-1]+1$, immediately follows from the recurrence equation in the naive algorithm, that is $D[i,\ j] = \min(D[i,\ j-1]+1, \dots)$. For the lower bound, $D[i\ , j-1]-1 \leq D[i,\ j]$, Figure 2.2 shows three possible cases of sequences of edit operations leading to the computation of $D[i,j]$. In the leftmost case $D[i\ , j] = D[i\ , j-1]+1$ and the lower bound holds. Middle case cannot happen, as the substitution shown in the rightmost case will always be cheaper than the one insertion and the one deletion in the middle case. The rightmost case showcases a sequence of one substitution followed by $k-1$ deletions (if $k=1$, there is only the substitution). $D[i\ , j]$ is then equal to either $D[i-k,\ j-1]+(k-1)+1$ if the substitution is a mismatch, or $D[i-k,\ j-1]+(k-1)$ if the substitution is a match, that is $D[i,\ j]+1 \geq D[i-k,\ j-1]+(k-1)+1$. However, we can always compute $D[i,\ j-1]$ by making k deletions from $D[i-k,\ j-1]$, that is $D[i,\ j-1] \leq D[i-k,\ j-1]+k$. By combining these equations we get $D[i,\ j]+1 \geq D[i,\ j-1]$, which means the lower bound holds.

This enables us to encode each element of vectors A and B (and A' and B') as the difference between itself and its predecessor. This is demonstrated in the third image in Figure 2.1. Furthermore, we can ignore K, as we are working with steps instead of values. As whatever value it is, it will not change the steps itself. For example, both the vectors $(2,1,1,2)$ and $(8,7,7,8)$ can be expressed as a step vector $(-1,0,1)$. A disadvantage is that obtaining the result from the computed matrix is slightly more complicated than just returning the last value from A' or B' in the bottom right corner. Nonetheless, we can just add up all the steps in vectors A in the first column of the matrix of blocks (which we already know is equal to m) and all the steps in vectors B' in the last row. Or vice versa for the first row with B and last column with A'.

2.2 Pseudocode

The pseudocode for the preprocessing step is shown in Algorithm 2. Let D_t be an integer matrix of size $(t_m + 1) \times (t_n + 1)$ used for computing A' and B'. Function STORE(…) saves A' and B' in the LUT so that they can be looked up later by supplying A, B, X and Y. Function COMPARISON is the same one as the one in the pseudocode for the naive algorithm.

Algorithm 2 The preprocessing step

```
for all A, B, X and Y do
    D_t[0, 0] ← 0
    for i ← 1 to t_m do                                  ▷ Decode A
        D_t[i, 0] ← D_t[i − 1, 0] + A[i − 1]
    for j ← 1 to t_n do                                  ▷ Decode B
        D_t[0, j] ← D_t[0, j − 1] + B[j − 1]
    for i ← 1 to t_m do                                  ▷ Compute the block
        for j ← 1 to t_n do
            tmp1 ← D_t[i − 1, j] + 1
            tmp2 ← D_t[i, j − 1] + 1
            tmp3 ← D_t[i − 1, j − 1] + COMPARISON(X[i], Y[j])
            D_t[i, j] ← min(tmp1, tmp2, tmp3)
    for i ← 1 to t_m do                                  ▷ Encode A'
        A'[i − 1] ← D_t[i, 0] − D_t[i − 1, 0]
    for j ← 1 to t_n do                                  ▷ Encode B'
        B'[i − 1] ← D_t[0, j] − D_t[0, j − 1]
    STORE(A, B, X, Y, A', B')
```

In the preprocessing step we go through $\mathcal{O}((3|\Sigma|)^{t_m+t_n})$ combinations of input values. Computing one block takes $\mathcal{O}(t_m t_n)$ time and the result takes up $\mathcal{O}(t_m + t_n)$ space. The time complexity is then $\mathcal{O}((3|\Sigma|)^{t_m+t_n} t_m t_n)$ and the LUT occupies $\mathcal{O}((3|\Sigma|)^{t_m+t_n}(t_m + t_n))$ memory.

The pseudocode for the computation step is revealed in Algorithm 3. Let's assume that m is divisible by t_m and n is divisible by t_n. Let S be a matrix of size $((m/t_m) + 1) \times ((n/t_n) + 1))$, where each element is an integer vector of size t_m (in other words A), and T a matrix of the same size, where each element is an integer vector of size t_n (in other words B). Function LOOKUP returns A' and B' from the LUT as if the inputs were A, B, X and Y.

Each element of S occupies $\mathcal{O}(t_m)$ space, and similarly each element of T occupies $\mathcal{O}(t_n)$ space. We spend $\mathcal{O}(1)$ time on each element (on the condition that each vector fits into a computer word), which leads us to time complexity of $\mathcal{O}(mn/(t_m t_n))$ and memory complexity of $\mathcal{O}(mn(t_m + t_n)/(t_m t_n))$.

Algorithm 3 The computation step

```
for i ← 1 to m/t_m do                                   ▷ Initialize S
    S[i, 0] ← (1, 1, ..., 1)
for j ← 1 to n/t_n do                                   ▷ Initialize T
    T[0, j] ← (1, 1, ..., 1)
for i ← 1 to m/t_m do                ▷ Fill S and T with values from the LUT
    for j ← 1 to n/t_n do
        (S[i, j], T[i, j]) ← Lookup(S[i − 1, j], T[i, j − 1],
            U[i × t_m ... (i + 1) × t_m], V[j × t_n ... (j + 1) × t_n])
result ← n                                              ▷ Add up the first row
for j ← 1 to n/t_n do                                   ▷ Add up the last column
    for k ← 0 to t_n − 1 do
        result ← result + T[m/t_m, j][k]
return result
```

Finally, if we assume $t_m = t_n = (\log_{3|\Sigma|} n)/2$, the total time complexity becomes

$$\mathcal{O}((3|\Sigma|)^{2(\log_{3|\Sigma|} n)/2}(\log^2_{3|\Sigma|} n)/4) + \mathcal{O}\left(\frac{mn}{(\log^2_{3|\Sigma|} n)/4}\right) =$$

$$= \mathcal{O}(n(\log^2_{3|\Sigma|} n)/4) + \mathcal{O}\left(\frac{mn}{(\log^2_{3|\Sigma|} n)/4}\right) =$$

$$= \mathcal{O}(n \log^2 n) + \mathcal{O}\left(\frac{mn}{\log^2 n}\right) = \mathcal{O}\left(\frac{mn}{\log^2 n}\right),$$

that is an asymptotic improvement over the naive algorithm. Similarly, the memory complexity will be $\mathcal{O}(n \log n) + \mathcal{O}(mn/\log n) = \mathcal{O}(mn/\log n)$.

2.3 Alphabet independent string encodings

In practice there arises a significant problem with the Four-Russians algorithm, which is that even modestly large Σ explode the size of the LUT. For example, if each element in the LUT occupies one byte, then for $t_m = 3$, $t_n = 4$ and $|\Sigma| = 4$ the size of the LUT is roughly 34 MB, but with $|\Sigma| = 26$ and the same t_m and t_n the size grows to almost 16 TB.

This section focuses on several encodings for X and Y that make the number of their combinations, and as a consequence the size of the LUT, independent of $|\Sigma|$. The general idea is that we only care whether two characters are the same or not (of course, this would not be the case if we required edit operations with varied costs for different characters).

2.3.1 First idea

First encoding, put forward by Kim et al. [4], is as follows: encode the first character of X as 1. Then iterate through the rest of X and Y, and if the given character has already been encoded before, encode it as the same value, and if not, encode it as the biggest yet encoded value plus one. For example, $X = bac$ and $Y = adab$ are encoded as $X_{enc} = 123$ and $Y_{enc} = 2421$.

The algorithm provided by the authors assumes that we have prepared beforehand an array Z of size $|\Sigma|$, which is indexable by the characters of Σ and has all values initialized to zero. Z is used for keeping track of the encoded values. For each character we first check Z, and if the value is not zero, we encode the character as whatever value is there, and if the value is zero, we encode it as the biggest yet encoded value (which we also keep in a temporary variable) plus one and store the value in Z. Afterward, we again iterate through X and Y and set the values in Z for the corresponding characters to zero, so that Z can be reused again. This approach has time complexity of $\mathcal{O}(t_m + t_n)$. The first element has one possible value, the second one two, the third one three and so on, which means the number of combinations is $(t_m + t_n)!$.

We can prepare such array even when $|\Sigma| = \infty$, as we can convert the input strings into strings over an alphabet of size $|\Sigma| = \mathcal{O}(m + n)$. We accomplish this by creating a new string S equal to the concatenation of U and V. Afterward, we sort it, remove duplicates and then for each character in U and V use binary search to locate their index in S, which is their value in the new alphabet. This takes us $\mathcal{O}((m+n)\log(m+n))$ time, which is then dominated by the running time of the computation step ($\mathcal{O}(mn/(\log^2 n))$).

2.3.2 Second idea

Second encoding was proposed by Bille and Farach-Colton [8]. The characters are encoded according to their position in a sorted array containing all characters appearing in both substrings. Characters present either in only X or in only Y are encoded as zero. For the same example, $X = bac$ and $Y = adab$ are encoded as $X_{enc} = 210$ and $Y_{enc} = 1012$. The algorithm given by the authors starts by iterating Y and using binary search to locate all the characters that are also present in X. The rest is similar to the algorithm in the last paragraph, that is the characters present in both strings are combined into a string S, sorted and have their duplicates removed. Then the encoded values of characters X and Y are equal to their position in S, and if they are missing in S, they are encoded as zero. We can achieve this in $\mathcal{O}((t_m + t_n)\log(t_m + t_n))$ time. Each element has $t_m + t_n + 1$ possible values, and as such there are $(t_m + t_n + 1)^{t_m+t_n}$ combinations.

2.3.3 Third idea

The encoding used in the implementation is a combination of both of these encodings. X is encoded according to the first encoding, afterwhich we iterate through Y and simply encode each character as the corresponding value in Z, which is either an already determined encoded value if the character is also present in X, or zero if not. Again, the same example, $X = bac$ and $Y = adab$ are encoded as $X_{enc} = 123$ and $Y_{enc} = 2021$. This approach has time complexity of $\mathcal{O}(t_m + t_n)$ and number of combinations equal to $(t_m)!(t_m+1)^{t_n}$. It is quite easy to see that this number is smaller or equal to the number of combinations generated by the first encoding (recall that t_m and t_n are positive integers):

$$\begin{aligned}
(t_m)!(t_m+1)^{t_n} &\leq (t_m+t_n)! \\
(t_m)!(t_m+1)^{t_n} &\leq (t_m)!(t_m+t_n)!/(t_m)! \\
(t_m+1)^{t_n} &\leq (t_m+t_n)!/(t_m)! \\
(t_m+1)\times(t_m+1)\times\cdots\times(t_m+1) &\leq (t_m+1)\times(t_m+2)\times\cdots\times(t_m+t_n)
\end{aligned}$$

and similarly is also smaller than the number of combinations generated by the second encoding:

$$(t_m)!(t_m+1)^{t_n} < (t_m+1)^{t_m+t_n} < (t_m+t_n+1)^{t_m+t_n}.$$

Another advantage is that encoding each Y is quite fast, and that each X has to be encoded only once for each row (this is also the case for the first encoding, however in this encoding reusing the alphabet is less cumbersome).

Of course, using this encoding slightly changes the aforementioned time and memory complexities. The LUT requires $\mathcal{O}(3^{t_m+t_n}t_m!(t_m+1)^{t_n}(t_m+t_n))$ memory and computing it takes $\mathcal{O}(3^{t_m+t_n}t_m!(t_m+1)^{t_n}t_mt_n)$ time. Also spending $\mathcal{O}(t_m+t_n)$ time on encoding vectors X and Y in each iteration of the computation step worsens the run time of computation step to $\mathcal{O}(mn(t_m+t_n)/(t_m\times t_n))$. After assigning $t_m = t_n = \log n$ we get $\mathcal{O}(mn/\log n)$. This can be averted by using another LUT, unfortunately this brings back the dependancy on $|\Sigma|$ to the size of the LUT and as such it is still an impractical solution for larger Σ.

2.4 Parallelization

This section in this chapter summarizes the possibilities for parallelization in the algorithm, as presented by Kim et al. [4]. It is easy to see that all the iterations in the for loop in the preprocessing step are independent of each other. For this reason, we can compute the LUT in $\mathcal{O}(t_mt_n)$ time with $3^{t_m+t_n}t_m!(t_m+1)^{t_n}$ threads.

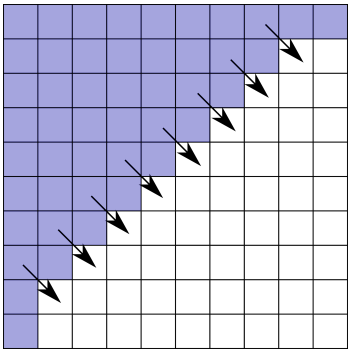

Figure 2.3: Parallel computation of the matrix in theory
Blocks with blue background are already computed. Black arrows represent threads working on new blocks.

The computation step is slightly trickier, as each block in matrices S and T is dependent upon the block in S that is right to the left of it and also upon the one in T right above it. However, this means that we can at least compute concurrently all the blocks on a diagonal, as is demonstrated in Figure 2.3. As each matrix has $m/t_m + n/t_n - 1$ diagonals, we can compute them in $\mathcal{O}((t_m + t_n)(m/t_m + n/t_n))$ time (which is equal to $\mathcal{O}(m + n)$ with $t_m = t_n = \log n$) using $\min(m/t_m, n/t_n)$ threads. This is however dominated by converting U and V to strings over an alphabet of size $\mathcal{O}(m+n)$, which takes us $\mathcal{O}((m + n) \log(m + n))$ time, nonetheless this task can also be parallelized to run in $\mathcal{O}(m + n)$ time.

2.5 Reducing search space

The idea used by the Ukkonen's algorithm [6] described in the previous chapter, which is that computing only a certain band of diagonals in the middle of the matrix is all that is required for the computation of edit distance, could also be used in the Four-Russians algorithm. Nonetheless, there are two issues pertaining to it.

The first one is that we would also have to save in the index to the LUT the information that tells us whether either A or B is outside the band of diagonals, thus increasing the number of combinations by a factor of 3 (not 4, since we skip the blocks where A and B are both outside the band of diagonals).

The second issue is that the matrices S and T would not be fully filled out and as such we would not be able to compute the edit distance by adding up all the steps in the last row to m (or by adding up all the steps in the last column to n). The solution is that we allocate a third matrix Q of size $((m/t_m) + 1) \times ((n/t_n) + 1)$, where each element is equal to the value in the

bottom right corner of a block, and we update this matrix together with S and T as we iterate through them. We do this by taking the value of the bottom right corner of the block above, which has already had to have been computed, and add to it all the steps of A', which takes us $\mathcal{O}(t_m)$ time, or $\mathcal{O}(1)$ time if we use yet another LUT. Then the resulting edit distance is equal to the bottom right corner of Q.

If m or n is not divisible by t_m or t_n, we pad U and V with p_m and p_n sentinel values until they are. These sentinel values can belong to Σ. Matrices Q, S and T are then of sizes $(((m+p_m)/t_m)+1) \times (((n+p_n)/t_n)+1)$. As the cell representing the edit distance between $U[1 \dots m]$ and $V[1 \dots n]$ has to be somewhere in the last block, we just stop the computation right before it and use the naive algorithm to compute it.

Chapter 3

Implementation Insights

3.1 Hardware and software used for testing

This chapter describes in detail an implementation of the algorithm introduced in the previous chapter. C++ is the programming language of choice, as it is well suited for performance intensive tasks and also possesses the support of the OpenMP technology, opening the door for easier parallelization of the algorithm.

The compiler used for compiling the implementation and the benchmarking code was gcc version 8.2.1 with `-O3` and `march=native` optimization flags. The processor used for running the code was Intel Xeon E5-2620 v2 with 6 physical cores, 12 logical cores, locked to a base frequency of 2.1 GHz. The cache sizes are as follows: L1I 12×32 KiB, L1D 12×32 KiB, L2 12×256 KiB and L3 2×15 MiB.

The inputs to the benchmarks were obtained by taking substrings from random positions of one of two files:

- Complete DNA sequence of *Escherichia Coli*. The file was processed by deleting the leading header line and removing all line breaks. This leaves us with 5 277 676 characters, but only 4 unique ones, with each letter representing a single nucleotide: A (adenine), C (cytosine), G (guanine) or T (thymine).
- *War and Peace* by Leo Tolstoy (translated to English). The introduction to the book was deleted, line breaks were removed and all the letters were converted to lowercase. The processed file contains 3 188 925 characters comprised of letters from the English alphabet, numbers, diacritics and, separators.

Each benchmark was repeated multiple times (depending on the benchmark) and the median value was chosen. The correctness of the implementation was tested by generating a considerable number of pairs of random

strings of different sizes from various alphabets, computing the edit distance between them, and verifying it matches up with a result from a battle-tested C++ library for computing edit distance – Edlib [9]. These tests were then repeated for several combinations of parameters t_m, t_n and various numbers of threads.

3.2 Main differences between theory and practice

Let's first summarize the main differences between the theory and the actual implementation.

To begin with, the parameters t_m and t_n are not equal to the log of m and n, but rather they are set to some constant value. This enables us to use the same LUT for all pairs of input strings and also improves the run time of the algorithm (since the $\mathcal{O}$ notation does not take into account such things as cache misses). The specific values of t_m and t_n are discussed in Section 3.4.

The LUT itself is just an array, and to be able to use A, B, X and Y as an index, we have to encode all of them into a single integer. This is discussed more in the following Section 3.3, but we also use this fact to generate all the combinations in the preprocessing step. Instead of using some sophisticated algorithm, the index variable is simply incremented and A, B, X and Y are decoded from it in each iteration. Preliminary benchmarks showed this to be the slightly slower solution, nonetheless, it was chosen for the simple fact that it is trivial to parallelize, as we will see in Section 3.5. As a side note, declaring arrays for these vectors before the loop itself and reusing them in each iteration instead of reinitializing them proved to be quite beneficial for performance.

As for the computation step itself, we employ the same optimization as in the Wagner-Fischer algorithm. That is, instead of initializing matrices S and T of size $((m/t_m) + 1) \times ((n/t_n) + 1)$, we only need vectors S and T of size (m/t_m) and (n/t_m), respectively. We could theoretically get by with only one of these, however, both of them are needed if neither m is divisible by t_m nor n is divisible by t_n, to reconstruct both the last row and the last column. They are also helpful when parallelizing the algorithm, as we will see in Section 3.5. The computation is also done in a column-major order instead of a row-major order. Benchmarks indicated it to be faster by about 1% and as a bonus it is consistent with [10]. The only difficulty this brings is that now Y is the first string to be encoded and X the second one, while in the previous chapter it was the other way around, which also means the number of their combinations has changed to $(t_n)!(t_n + 1)^{t_m}$.

The idea of computing only a band of diagonals in the middle was not implemented, as it was felt that such a heuristic would obscure the difference in performance between different algorithms.

Lastly, instead of converting U and V to an alphabet of size $\mathcal{O}(m+n)$ we use a shortcut and just declare an array of size 256 that is able to be indexed by all the values of the `char` type. Of course, the unfortunate consequence is that $|\Sigma|$ is limited to 256.

3.3 Packing values into bits

Let's consider the problem of packing an array of n elements into a single integer. Trivially, if an element has a particular number of possible values (V), the number of bits (b) needed to encode this element is $b = \lceil \log_2 V \rceil$. To get b for the whole array, we just need to add up b of all the elements.

Ofcourse, if V of any element is not a power of two the array will not be packed optimally, as some values of the integer are going to be skipped. Let's define V of the first element as V_1, V of the second element as V_2 and so on, and likewise for b. Then we can get the memory efficiency, that is the ratio describing how efficiently the array is packed, like this:

$$\frac{\text{used values}}{\text{all values}} = \frac{V_1 \times V_2 \times \cdots \times V_n}{2^{b_1} \times 2^{b_2} \times \cdots \times 2^{b_n}}$$

When all V are not a power of two, memory efficiency deteriorates quite quickly. For example, with $V = 3$ and $n = 2$, it is equal to 56%, with the same V and $n = 4$ it's 32%, and with $n = 6$ it's only 18%. For this reason, we are going to discuss another way of packing such arrays, as shown by Mikkelsen [10], and flesh it out to achieve 100% memory efficiency.

The main idea is to pack the values in a different base than base two. We just have to use the arithmetic operations applicable for any base, instead of the bitwise operations applicable because of a certain internal representation. Left shift by $x \times b$ bits is replaced with multiplication by V^x, right shift by $x \times b$ bits is replaced with division by V^x and bitwise AND of the last $x \times b$ bits is replaced with modulo V^x. For example, let's say that we have an array $(1,\ 2,\ 0,\ 2)$ and $V = 3$. To encode the array in base three, we start with zero, then we iterate the array and at each iteration we multiply what we have so far by three and add the value to it. For the example array it looks like this:

- $0 \times 3 + 1 = 1$
- $1 \times 3 + 2 = 5$
- $5 \times 3 + 0 = 15$
- $15 \times 3 + 2 = 47$

and the result is 47, that is $1 \times 27 + 2 \times 9 + 0 \times 3 + 2 \times 1$.

To decode the array, we access the last element with modulo three, and afterward remove it with division by three. Again, for the example array it looks like this:

- $47\%3 = 2$ and $47/3 = 15$
- $15\%3 = 0$ and $15/3 = 5$
- $5\%3 = 2$ and $5/3 = 1$
- $1\%3 = 1$

and we get the same array, (1, 2, 0, 2) (the fact that it's backward can be circumvented in the same way as when we are encoding in base two). The advantage of doing all this is that when the base is equal to V the values are packed tightly and there aren't any intermediary unused values. The number of bits needed is then $\lceil \log_2 V^n \rceil$. Differences for $V = 3$ and various n can be seen in Table 3.1.

$V = 3,\ n =$	1	2	3	4	5	6	7	8
b for base2	2	4	6	8	10	12	14	16
b for base3	2	4	5	7	8	10	12	13

Table 3.1: Saved bits with the base3 encoding

We can make use of this when we are encoding A ($V = 3$), B ($V = 3$) and X ($V = t_n + 1$). However, Y has different V for each of its elements. Nonetheless, it turns out that this approach still works wonders. We just have to use V_i instead of V when encoding/decoding. For example, if we have array (2, 1, 2, 0) with $V_1 = 5$, $V_2 = 2$, $V_3 = 3$ and $V_4 = 4$, the encoded result is $2 \times (2 \times 3 \times 4) + 1 \times (3 \times 4) + 2 \times (4) + 0 = 64$. This means that we can encode all four vectors with this approach, but not only can we encode each vector as a number and then put them next to each other, we can encode all of them as a single number, thus eliminating any memory inefficiency (last range of unused values can be skipped, as the size of the LUT itself does not have to be a power of two). How much memory is saved when allocating the LUT is demonstrated in Table 3.2.

To be more precise, memory inefficiency will not be quite zero, as there will still be a lot of unused values in the integer used for indexing. That is because the number of combinations of Y is not $t_n!$, but $\mathcal{O}(t_n!)$. What is meant by that, is that for example the sequence 0, 1, 1, 3 can never happen, as the algorithm would make the third element 2 and not 3, but it is still a valid value, since $0 \times (2 \times 3 \times 4) + 1 \times (3 \times 4) + 1 \times (4) + 3 = 19$. However, an encoding that would encode Y in such a way that these combinations would be skipped and at the same time would not be horribly slow was not discovered, and as such not implemented.

To be even more precise, the elements itself are not perfectly allocated in the LUT. Each vector A' (or B') takes one whole byte if $t_m \leq 5$ (or $t_n \leq 5$), or even two bytes if $t_m > 5$ (or $t_n > 5$), even though they don't need all

t_m/t_n	standard	modulo
1/1	54 B	36 B
2/2	13.65 kB	2.92 kB
3/3	3.67 MB	559.87 kB
4/4	14.80 GB	196.83 MB
5/5	30.33 TB	110.20 GB

Table 3.2: Size of the LUT with the standard and the modulo encoding
One element in the LUT occupies two bytes.

that space. This issue can be alleviated by encoding A' and B' into one integer. This reduces the memory consumption for some values of t_m and t_n and increases it for others. As it did not make much of difference, this was not implemented in the final version of the implementation. Preliminary benchmarks showed that it causes a negligible slowdown in the computation step if bitwise operations are used to separate them and a very significant slowdown if modulo arithmetic is used to separate them.

This brings us to the question of speed, as multiplication, division, and modulo are without a doubt slower instructions than bitwise operations. However, we only need to encode A and B when we are storing them in the LUT and decode them only when we are adding up all the values in the last row of the matrix (and also at the beginning of the preprocessing step). This means that the hottest part of the algorithm, the loop in the computation step, is not affected by it. On the other side, X and Y are also decoded in the preprocessing step, but Y is encoded at the start of each column in the computation step and X even during each iteration. Nonetheless, preliminary benchmarks did not indicate any significant performance loss, as any possible effect was perhaps offset by the smaller and improved layout of the values in the LUT. Even for the preprocessing step, individual iterations were about 10% slower, but the overall running time was still faster, as the number of iterations decreased significantly.

Now, for the order of bits of A, B, X and Y in the index. The answer to this question depends on whether the computation is done in row-major or column-major order. That is because with row-major order X stays the same during the computation of the whole row. Same goes for column-major order and Y. Now, the reason why this is beneficial is that the accesses to the LUT are kept in roughly the same area, leading to fewer cache misses. As column-major order was used in the implementation, Y occupies the most significant bits in the index. As for the rest, preliminary benchmarks proved that best performance comes from X next, then A and lastly B (though this is dependant on the values of t_m and t_n), that is the ordering should be $YXAB$, with Y occupying the most significant bits and B the least.

3.4 Choosing t_m and t_n

Choosing the optimal values for t_m and t_n is a nontrivial issue, but a crucial one for a good performance of the algorithm. On one hand, bigger values directly decrease the number of iterations in the computation step, on the other hand, they also increase the size of the LUT and incurring a recurring amount of cache misses slows down the algorithm quite a bit. Furthermore, bigger values of t_n lead to larger Y, which, as was discussed in the previous paragraph, leads to better cache locality. However, as the number of values in each element of X is directly tied to t_n, it being small works wonders for reducing the size of the LUT.

Extensive benchmarking showed several pairs of t_m and t_n being competitive, nonetheless the results are reliant on several factors:

- The size of the CPU cache – larger cache allows the efficient use of larger tables, which means larger tm or t_n, which leads to better speedup. However, even we if focus only on the tested machine, there are more things to consider, such as
- The size of the input strings – even though t_m and t_n are constant, increasing the size of the input strings leads to increased speedup. One possible explanation is that longer columns allow Y to stay the same for longer periods of time, leading to better cache locality. However, the rate of speedup is different for each individually sized LUT and therefore for different values of t_m and t_n.
- The number of threads used – this factor will be discussed in the following section.
- The content of the input strings – the LUT is divided into parts based upon whether letters in Y repeat itself or not. Of course, even though letters from the English alphabet are not distributed uniformly, repeated characters occur much less in *War and Peace* than in the genome of *Escherichia coli*. This leads to certain parts of the LUT (or even only a certain part) being heavily overutilized, while the others are heavily underutilized. Which is actually useful, as this leads to better cache locality and bigger values of t_m and t_n can be chosen and so on. This is illustrated in Figure 3.1. We can see that in the case of the bacteria, no part of the table is accessed more than 10% of the time, while in the literary work the part where every element of Y is unique is accessed more than 80% of the time!

All of this means that there is not a single optimal pair of values for t_m and t_n, even when only the one CPU on the testing machine is taken into account. As a result, the values chosen for parameters t_m and t_n in this thesis differ from benchmark to benchmark.

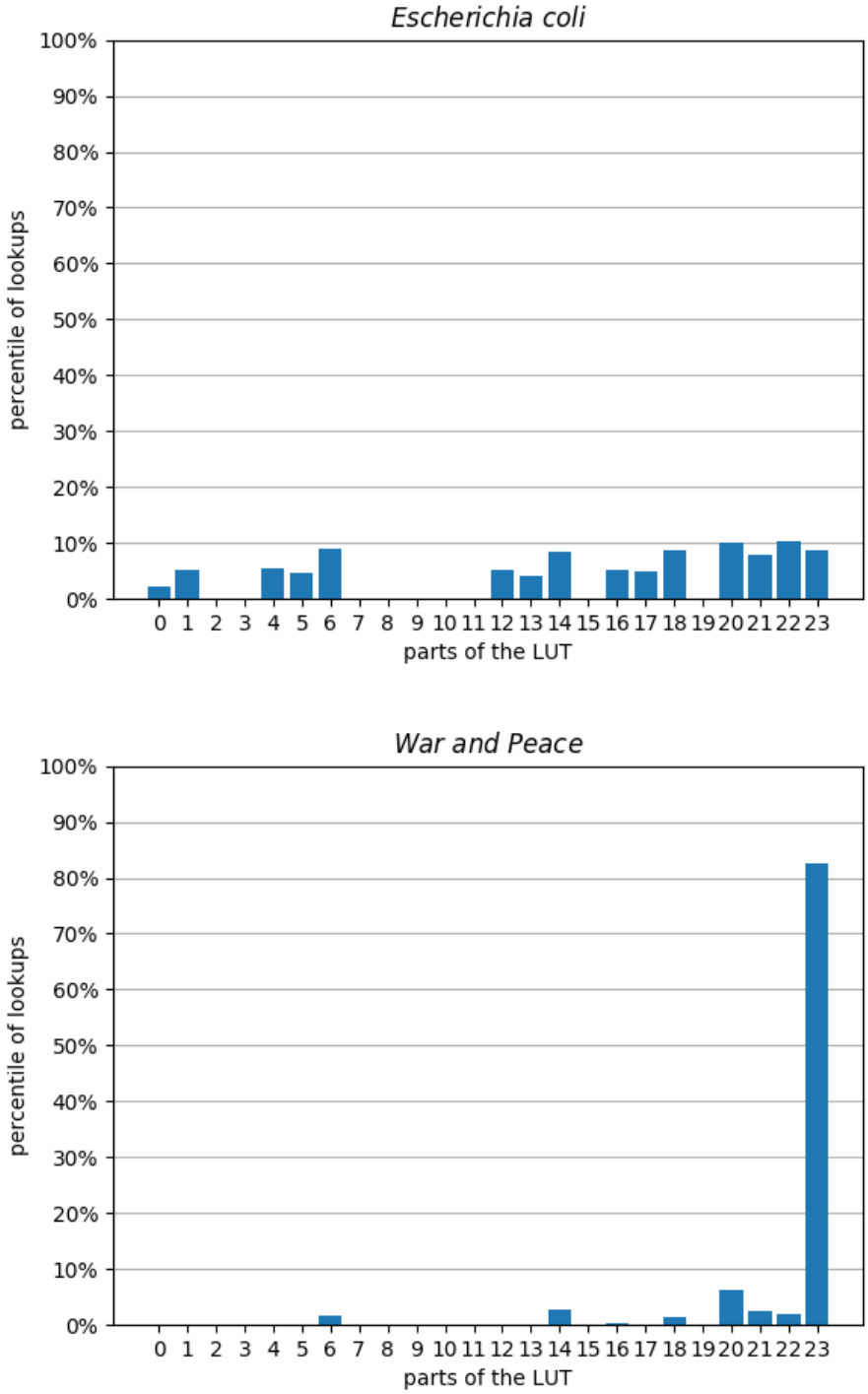

Figure 3.1: LUT access patterns

The parameter t_n is equal to 4. This means that there are $4! = 24$ combinations of Y and the corresponding LUT was divided into 24 parts depending on the value of Y. $Y = 0$ means all elements in Y are the same, $Y = 23$ means all elements are unique. Several values of Y, more specifically 2, 3, 7, 8, 9, 10, 11, 15 and 19, are wasted – they represent combinations which can never happen, as is described in the end of Section 3.3.

In the implementation, t_m and t_n are configurable, however, they have to be provided at compile time, that is, as template arguments. The reason for that is that setting these values at compile time enables the compiler to perform more optimizations. Examples include unrolling loops where the parameters are used as loop indices or replacing expensive operations, such as modulo or division, with simpler ones. Preliminary benchmarks have shown that providing these parameters at runtime slows down the preprocessing step by about 50-200% and the computation step by about 5-30%. Another possible benefit of templates is that the various types can be chosen such that they are the smallest possible types that can contain all the needed values for each pair of t_m and t_n. However, making the table index `uint32_t` and the integers representing A and B `uint8_t` is enough to represent all reasonable values of t_m and t_n, so this feature is not as useful. What is definitely not useful is the fact that implementation for templated functions has to be in the header, and as such it is compiled every time a translation unit including this header is compiled.

3.5 Parallelization

As the parallelization in the implementation is done with the help of the OpenMP API, we will start with a short introduction of it. OpenMP's aim is to enable shared memory parallel programming without the programmer actually having to program the interactions between the threads. It is a set of compiler pragmas (also called directives) (and also some subroutines) that when placed in the correct spots in the source code (for example, before a for loop) instruct the compiler to parallelize the code by itself. Without a compiler flag, the directives are ignored, and as such the same source code can be used for both the sequential and the parallel version.

Now, recall that the only input to each iteration of the preprocessing step is the loop index from which A, B, X, and Y are decoded. This means that the calculations in each iteration are totally independent of each other (except for the fact that the results are saved in the same LUT) and the preprocessing step can be parallelized with a single OpenMP pragma, such as this one:

```
...
#pragma omp parallel for private(Y, X, A, B)
for (size_t YXAB = 0; YXAB < LUT_size; ++YXAB) {
...
```

The `private` clause is needed for the vectors, as they are in reality used as local variables, just declared before the loop for performance reasons. Thanks to the efficient memory encoding each (most) loop index is a valid encoded combination of A, B, X and Y and each iteration takes around the same time. For this reason, the default static schedule is a reasonable choice. As can be seen in Table 3.3, performance scales with additional threads quite

threads	speedup
2	1.98
3	2.85
4	3.79
5	4.54
6	5.43
7	6.34
8	7.25
9	8.15
10	9.06
11	9.96
12	10.84

Table 3.3: Scalability of the preprocessing step
Parameters are $t_m = 3$, $t_n = 4$.

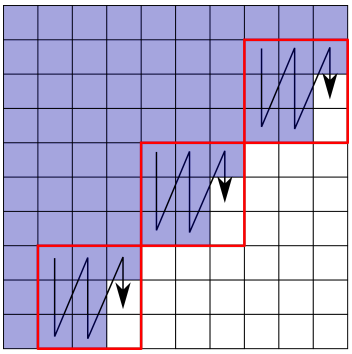

Figure 3.2: Parallel computation of the matrix in practice
Blocks with blue background are already computed. Black arrows represent threads working on new blocks. Red outlines show the borders of chunks.

well. However, it should be noted that for different values of t_m and t_n the LUT can be very small and the overhead of creating the threads itself (and also possibly the problem of false sharing) can dominate the running time.

Let's move on to the computation step. As we have seen in the previous chapter, all the blocks on each diagonal of S and T can be computed concurrently. Nonetheless, similarly to the naive algorithm, as discussed by Bednárek et al. [11], computing one block takes a very small amount of time and the communication overhead offsets any possible performance gains. This means that we need to parallelize at a coarser level and so S and T are split into chunks of a certain size. Afterward, instead of computing diagonals of individual blocks, diagonals of chunks are computed. Figure 3.2 illustrates the problem.

Preliminary benchmarks indicated that chunks of size 64×64 are a good unit of work to base the algorithm around. However, this means that the input strings have to be quite large before parallelization can be utilized. For example, when $|S| = |T| = 2^{10}$ there are only nine diagonals of chunks, with only three of these being of size four and the rest being even smaller. Nonetheless, setting the size of chunks to smaller values did not seem to improve the run time much, probably due to the overhead of organizing the threads. Setting it to larger values proved fruitful for larger $|S|$ and $|T|$, however, since making an equation that works well in most cases proved to be quite difficult, chunks were set to size 64×64 for simplicity's sake.

Even if the size of the input strings is kept high, there are few more important factors, which are apparent in Table 3.4. As we can see, the speedup is very good for $t_m = t_n = 1$, which is unfortunately not that helpful, as with these parameters the algorithm is very slow. As the size of the LUT grows the gains begin to diminish. Some of this could be caused by higher values of t_m and t_n having a smaller amount of blocks and thus a smaller amount of chunks. Nonetheless, it seems that bigger impact was caused by the threads sharing the space of the L3 cache or similarly by the small size of chunks leading to often changing values of Y and thus worse cache locality. This is demonstrated by the fact that the more cache-friendly *War and Peace* improved the speedup for larger t_m and t_n quite a bit.

Finally, the other half of the computation step, that is the computation of the edit distance from S and T after the main loop has finished, took up $<1\%$ of the run time in preliminary benchmarks, and as such parallelizing it was deemed unnecessary.

3.5.1 SIMD

Another speedup could be theoretically gained by using SIMD (Single Instruction Multiple Data) parallelism. That is using specialized vectors registers (of sizes 64, 128, 256 or 512 bits) which can contain multiple values at once and using specialized vector instruction sets (such as SSE or AVX) that can operate on all these values at the same time.

Bednárek et al. [11] have shown that the naive algorithm can be parallelized this way and the same approach can theoretically be used for Four-Russians. While we cannot simply compute multiple rows (or columns) at once due to their dependency on each other, we can compute them if they are slightly "skewed", as is sketched in Figure 3.3. This makes the implementation slightly more complicated since a few blocks at the start and at the end have to be computed in scalar fashion. However, the big problem is that vectorized instructions are not well suited for looking up values in different places in a LUT (if the LUT is larger than the size of the vector register, which it almost definitely is in our case) and as such this approach was not pursued in the implementation.

threads	1/1	2/3	3/4	4/4
1	21568	3596	2470	4567
2	2.08	1.78	1.02	1.25
3	3.02	2.55	1.27	1.18
4	3.99	3.33	1.64	2.02
5	4.81	4.04	1.96	1.95
6	5.76	4.84	2.28	2.49
7	6.7	5.59	2.65	2.87
8	7.65	6.37	3.0	3.11
9	8.58	7.13	3.38	3.35
10	9.52	7.77	3.7	3.6
11	10.44	8.45	4.09	3.89
12	11.37	8.98	4.42	4.51

threads	1/1	2/3	3/4	4/4
1	21558	3598	2178	2214
2	2.08	1.81	1.48	1.4
3	3.02	2.59	2.01	1.88
4	3.99	3.43	2.62	2.38
5	4.82	4.12	3.1	2.8
6	5.77	4.91	3.67	3.26
7	6.72	5.71	4.25	3.86
8	7.66	6.49	4.85	4.25
9	8.62	7.26	5.44	4.96
10	9.54	7.96	5.92	5.33
11	10.46	8.67	6.53	5.98
12	11.38	9.38	7.08	6.48

Table 3.4: Scalability of the computation step
The first table is *Escherichia coli*, the second one is *War and Peace*. The size of the input strings is equal to 65 536. Values in the headers of columns are the values of parameters t_m and t_n. Values in the first row represent the run time in milliseconds for the sequential version, while the other rows represent the speedup gained with parallelization.

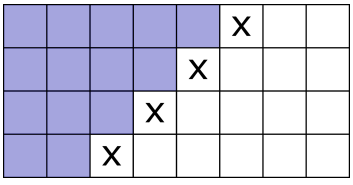

Figure 3.3: Utilizing SIMD instructions by "skewing" rows
Blocks with blue background are already computed. Black crosses show which blocks are computed with a single vector operation.

Chapter 4

Experimental Evaluation

In this final chapter, we are going to evaluate the Four-Russians algorithm (from now on referred to as FR) by comparing it to two popular algorithms for computing edit distance. The hardware and software configuration used was described in Section 3.1 in the previous chapter. The main competitor is Edlib, a C/C++/Python library by Šošić and Šikić [9] focused on computing edit distance and also various problems concerning sequence alignment in bioinformatics. The library implements the Myers's bit-vector algorithm [7] (from now on referred to as MVB) repurposed for computing edit distance. As mentioned in the last section of the first chapter, this algorithm computes up to w elements at time (where w is the size of the computer word, which is 64 on the testing machine) by representing columns of the dynamic programming matrix as vectors of bits and then using clever combinations of bitwise operations to swiftly compute them.

The time complexity is $\mathcal{O}(kn/w)$ where k is some parameter, and if the resulting edit distance is bigger than k, the algorithm instead reports an invalid result. The solution chosen for this in Edlib is the same one as in the Ukkonen's algorithm, which is that k is set to a certain value (64) and it is doubled every time the algorithm fails to find a solution until it finally finds one. In these benchmarks, k was always set to m. Due to the nature of the input strings this actually improved the performance.

To provide a baseline for both of the algorithms the naive algorithm was also implemented. Each implementation was pursuing the same task – computing edit distance, without saving the edit sequence or similar tasks. The benchmarks for FR "cheat" a little bit, as they do not include the time required for the preprocessing step. However, as the preprocessing step has to be run only once for each combination of t_m and t_n, it can be imagined that each benchmark was instead run so many times that the overhead of the preprocessing step was reduced to insignificance. The experiments are split into three sections with regards to the sizes of input: when both m and n are small, when only m is small and n is large, and when both m and n are large.

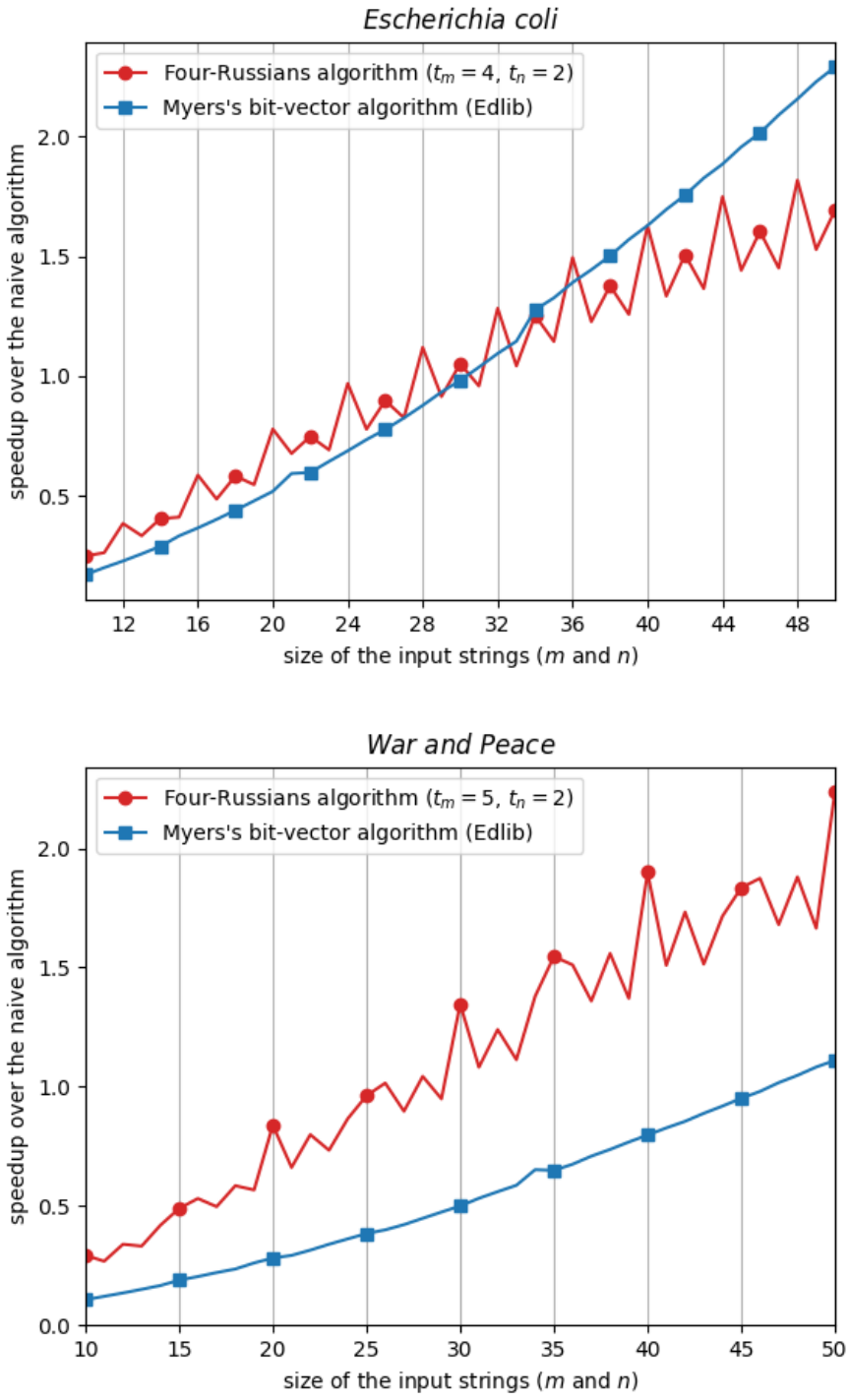

Figure 4.1: Comparison of run times for small m and n

4.1 Small m and n

The results for $m = n \geq 10$ and $m = n \leq 50$ can be seen in Figure 4.1. At first glance, the jagged look of the curve representing FR looks quite interesting. Initially, it might seem that this is due to the fact that accesses to the lookup table take an uneven amount of time and that there is not enough of them (due to the small input sizes) to average it out. However, the tests for each m and n are run many times, the lookup table at least for parameters $t_m = 4$ and $t_n = 2$ is small enough (236 kB) that after just being constructed it should still occupy lower levels of cache and mainly the irregularities displayed are actually quite regular. The real reason is that when m and n are not clearly divided by t_m and t_n, the rest of the dynamic programming matrix has to be computed with the naive algorithm. This can be demonstrated by the fact that the speedup is largest for values of 12, 16, 24 and so on (or 10, 15, 20 and so on), that is when the input sizes were divisible by both t_m and t_n. In contrast, the curve representing MVB is very linear, as is expected (except for a curious blip at around 34).

On a different note, the results indicate that the constants disregarded in the asymptotic analysis are higher for MVB. The slower startup could be caused by some peculiarities of Edlib, but since the library was built foremost for speed, that does not seem probable. It is more likely caused by the fact that MVB does not achieve its real speedup until m is at least 64, as we will see in the next section. In the case of *Escherichia coli*, by the time FR overtakes the naive algorithm, MVB is already speeding up at a faster rate. However, the roles are reversed for *War and Peace*, where FR achieves similar times, but MVB gets much slower (we can compare the results from both cases as the time taken by the naive algorithm is almost the same for both of them). It is unclear why MVB slows down so much, but we will see in the last section that it catches up later.

4.2 Small m and large n

This section concerns itself with the case, where m is very small relative to n, as is usually the case in approximate string matching, which is what MVB was designed for. Nonetheless, as can be witnessed in Figure 4.2, FR is actually faster when $m \leq 32$ for *Escherichia coli*, and when $m \leq 44$ for *War and Peace*. The reason for that is that MVB computes up to $w = 64$ elements at once, but only in a single column. If $m < w$, it can only compute up to m elements at once (and this effect did not seem to be alleviated by simply switching m and n). This is demonstrated by the fact that for all values of m the run time for MVB is almost equal. English text improved the run time of FR for reasons described in the implementation chapter, however, this time the difference in the content of the strings did not seem to change the run time of MVB much.

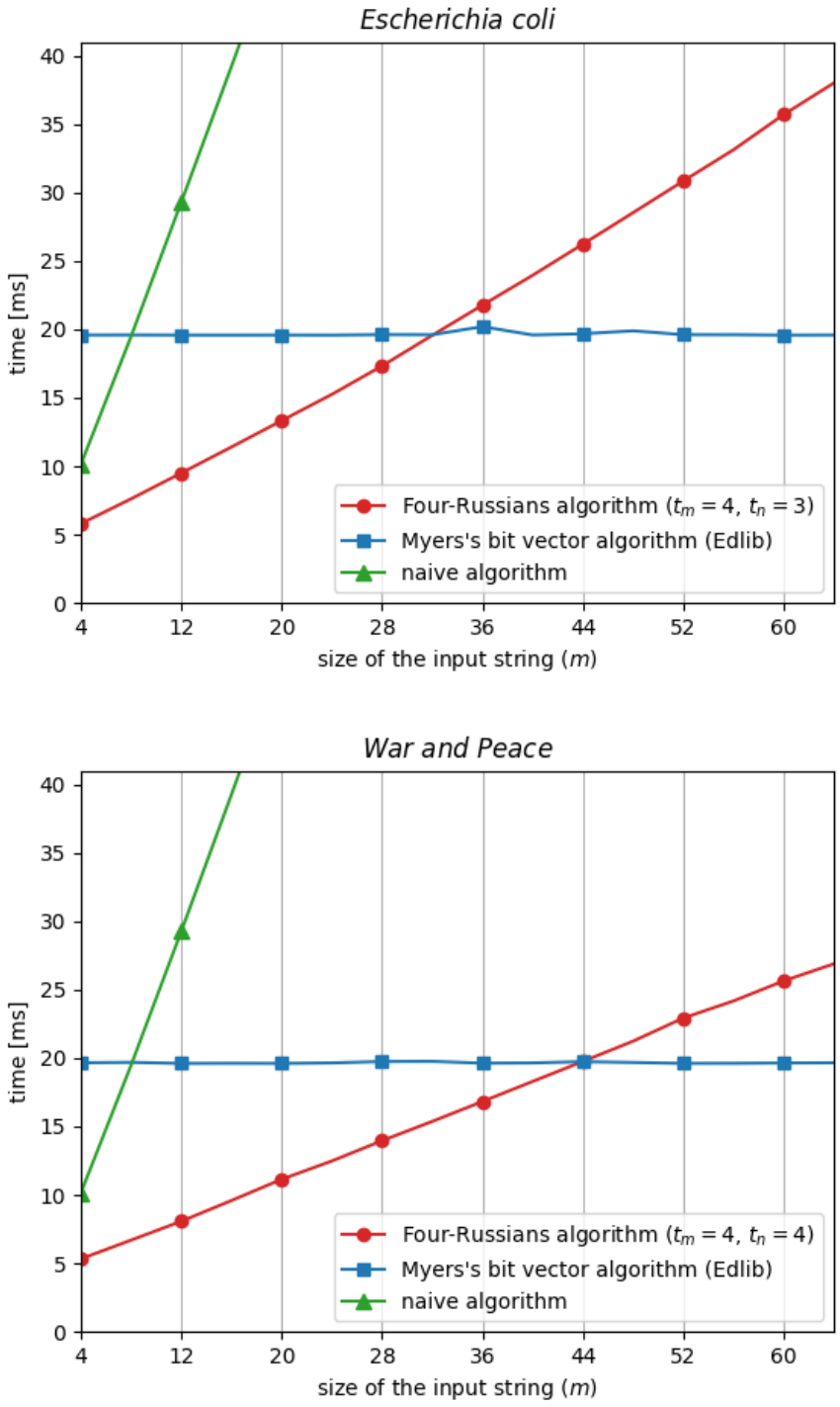

Figure 4.2: Comparison of run times for small m and large n $n = 1048576$.

4.3 Large m and n

The results for $m = n \geq 2^6$ and $m = n \leq 2^{18}$ can be seen in Figure 4.3. Both a single-threaded and a multi-threaded version of FR were benchmarked. The multi-threaded variant is always set to max threads, which is 12 on the testing machine, and the overhead of the threads actually makes it slower than the single-threaded for lower m and n. On the other hand, when the sizes of the input strings get large enough to make use of the parallelization the speedup is dramatically improved.

Even though the situation is markedly better when the algorithm is working on the historical novel, it is is still outmatched by the more modern algorithm. While FR has lower time complexity of $\mathcal{O}(mn/\log m \log n)$, it is of no use as in practice the logarithms have to be substituted by constants. It would require blocks of size 8×8 to match the theoretical speedup of MVB (or more precisely something similar to 1×64, since $\mathcal{O}(t_m)$ time is spent on encoding X at each iteration), and the LUT for these blocks would take around 3.7 exabytes of memory. Even if infinite memory were available, it would also be needed for the extraordinary large input strings, as a probable cache miss in each iteration of the algorithm would make the constant hidden by the big $\mathcal{O}$ analysis very noticeable. When we put it all together, it is not surprising that MVB is faster, as it computes up to 64 elements at a time using bitwise operations, while FR computes only up to around 20 elements using a LUT.

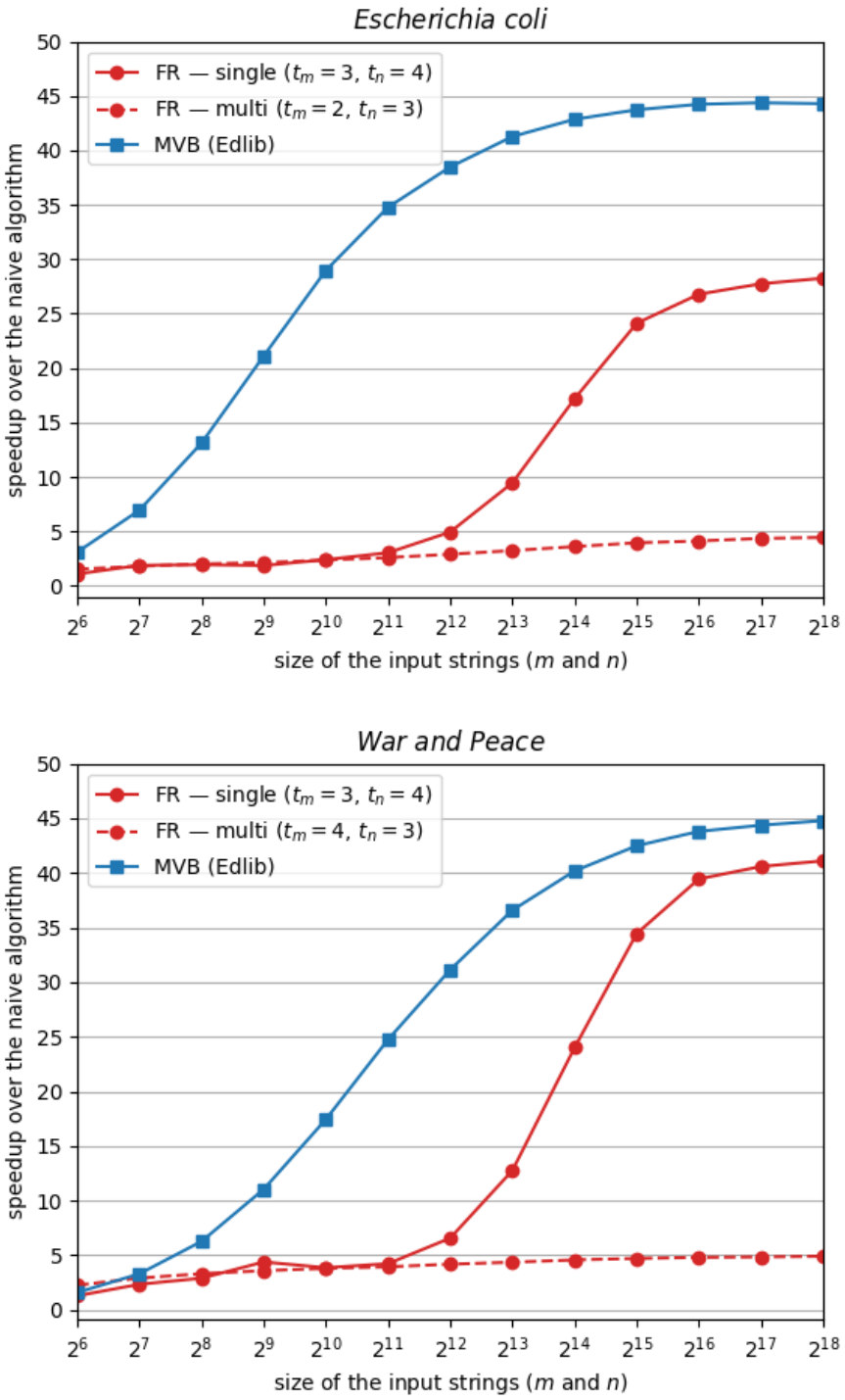

Figure 4.3: Comparison of run times for large m and n

Conclusion

In this thesis the Four-Russians algorithm for computing edit distance was analyzed, its implementation details were discussed and its performance was evaluated.

One of the most interesting things about the algorithm is trying to optimize the size of the lookup table and similar efforts were also a major part of this thesis. A slight improvement upon the string encoding by Kim et al. [4] was introduced, further reducing the number of combinations of the substrings of size t ($t = t_m = t_n$) from $\mathcal{O}((2t)!)$ to $\mathcal{O}(t!(t+1)^t)$, while also taking less time to be computed.

What's more, it was tested that using modulo arithmetic for packing values into bits, as mentioned by Mikkelsen [10], makes the layout of the lookup table significantly more efficient and improves the overall performance of the algorithm in most cases. It would be interesting to see how much more can the size of the lookup table be reduced in practice, especially by utilizing the work done by Brubach and Ghurye [12]. In addition to all this, a simple way to practically parallelize the algorithm was also explained, which results in up to 8× speedup with 12 cores.

Continuing with the topic of performance, the implementation of the Four-Russians algorithm did quite well when the size of at least one of the input strings was kept small (< 64) and was otherwise quickly outmatched by the Myer's bit-vector algorithm implemented by Edlib. However, the problem with these small strings is that while the preparation of the lookup table does not take a long time, it has to either be done beforehand and then kept around for when it is needed, or the values of t_m and t_n have to be chosen carefully so that the construction takes an insignificant amount of time.

This brings us to the second issue, which is how reliant the performance of the algorithm is on these parameters, for with suboptimal values the algorithm might be up to ten times slower, or possibly even more. However, we have seen that the optimal values depend not only on internal factors, such as the layout of the index to the lookup table or the encodings chosen for the individual

parts of the index, but also on several outside factors: the size of the input strings, the content of the input strings, the size of the processor cache and the number of threads. All these factors together make it quite difficult to configure the algorithm optimally.

To summarize, it seems that the Four-Russians algorithm is more interesting from a theoretical than a practical standpoint, at least when it comes to computing edit distance.

Bibliography

1. ARLAZAROV, V. L.; DINITZ, Y. A.; KRONROD, M. A.; FARADZHEV, I. A. On economical construction of the transitive closure of an oriented graph. In: *Dokl. Akad. Nauk SSSR, 194*. 1970, pp. 487–488. [in Russian]. English translation. In: *Soviet Math. Dokl.*, 11. 1970, pp. 1209-1210.
2. MASEK, William J.; PATERSON, Michael S. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*. 1980, vol. 20, no. 1, pp. 18–31. ISSN 0022-0000. Available from DOI: 10.1016/0022-0000(80)90002-1.
3. BACKURS, Arturs; INDYK, Piotr. Edit Distance Cannot Be Computed in Strongly Subquadratic Time (Unless SETH is False). In: *Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing*. Portland, Oregon, USA: ACM, 2015, pp. 51–58. STOC '15. ISBN 978-1-4503-3536-2. Available from DOI: 10.1145/2746539.2746612.
4. KIM, Youngho; NA, Joong Chae; PARK, Heejin; SIM, Jeong Seop. A space-efficient alphabet-independent Four-Russians' lookup table and a multithreaded Four-Russians' edit distance algorithm. *Theoretical Computer Science*. 2016, vol. 656, pp. 173–179. ISSN 0304-3975. Available from DOI: 10.1016/j.tcs.2016.04.028. Stringology: In Celebration of Bill Smyth's 80th Birthday.
5. WAGNER, Robert A.; FISCHER, Michael J. The String-to-String Correction Problem. *J. ACM*. 1974, vol. 21, no. 1, pp. 168–173. ISSN 0004-5411. Available from DOI: 10.1145/321796.321811.
6. UKKONEN, Esko. Algorithms for approximate string matching. *Information and Control*. 1985, vol. 64, no. 1, pp. 100–118. ISSN 0019-9958. Available from DOI: https://doi.org/10.1016/S0019-9958(85)80046-2. International Conference on Foundations of Computation Theory.

7. MYERS, Gene. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In: FARACH-COLTON, Martin (ed.). *Combinatorial Pattern Matching*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–13. ISBN 978-3-540-69054-2. Available from DOI: 10.1145/316542.316550.

8. BILLE, Philip; FARACH-COLTON, Martin. Fast and compact regular expression matching. *Theoretical Computer Science*. 2008, vol. 409, no. 3, pp. 486–496. ISSN 0304-3975. Available from DOI: https://doi.org/10.1016/j.tcs.2008.08.042.

9. ŠOŠIĆ, Martin; ŠIKIĆ, Mile. Edlib: a C/C ++ library for fast, exact sequence alignment using edit distance. *Bioinformatics*. 2017, vol. 33, no. 9, pp. 1394–1395. ISSN 1367-4803. Available from DOI: 10.1093/bioinformatics/btw753.

10. MIKKELSEN, Anders Høst. *Local alignment using the Four-Russians technique*. Aarhus, 2015. Available also from: http://cs.au.dk/~cstorm/students/HoestMikkelsen_Jun2015.pdf. Magister's thesis. Aarhus University, Department of Computer Science. Advisor: Christian Nørgaard Storm Pedersen.

11. BEDNÁREK, David; BRABEC, Michal; KRULIŠ, Martin. On Parallel Evaluation of Matrix-Based Dynamic Programming Algorithms. In: 2015.

12. BRUBACH, Brian; GHURYE, Jay. A Succinct Four Russians Speedup for Edit Distance Computation and One-against-many Banded Alignment. In: NAVARRO, Gonzalo; SANKOFF, David; ZHU, Binhai (eds.). *Annual Symposium on Combinatorial Pattern Matching (CPM 2018)*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, vol. 105, 13:1–13:12. Leibniz International Proceedings in Informatics (LIPIcs). ISBN 978-3-95977-074-3. ISSN 1868-8969. Available from DOI: 10.4230/LIPIcs.CPM.2018.13.

Appendix A

List of Symbols

U and V	input strings
m and n	lengths of the respective input strings
Σ	alphabet from which the input strings are from
D	dynamic programming matrix used for computing edit distance, $D[i,\ j]$ is equal to the edit distance between the substrings $U[1 \ldots i]$ and $V[1 \ldots j]$
k	a parameter that represents a cut-off point under which edit distance is not computed
w	size of a computer word
t_m and t_n	parameters for the size of one block
K	the top left corner of a block
A	the first column of a block without the first element
B	the first row of a block without the first element
A'	the last column of a block without the first element
B'	the last row of a block without the first element
X	the substring of U corresponding to a block
Y	the substring of V corresponding to a block
S	either a matrix or a vector where each element is A
T	either a matrix or a vector where each element is B
LUT	lookup table
FR	Four-Russians (algorithm)
MBV	Myers's bit-vector (algorithm)

Appendix B

Contents of the Enclosed CD

```
root/
├── bin/ ..... executable files (after they are built)
├── BP_Rejmon_Martin_2019.pdf ..... this thesis (in the PDF format)
├── CMakeLists.txt ..... configuration file for CMake
├── input/ ..... input files for the benchmarks
├── latex/ ..... LaTeX source for the thesis
├── README.txt ..... build instructions
└── src/ ..... C++ source code for the implementation
```