Local alignment using the Four-Russians technique
Anders Høst Mikkelsen, 20106624

Master’s Thesis, Computer Science
June 2015
Advisor: Christian Nørgaard Storm Pedersen
Abstract

Four-Russians is a method that can be used to improve the execution time of certain algorithms with a factor $\log^2(n)$ by precomputing common subproblems. Dan Gusfield adapted the original Four-Russians method to compute unit cost edit distances (a form of sequence alignment) in time $O(n^2/\log^2(n))$. "Sequence" is a general term, and can contain many kinds of data. DNA, text strings, and so on. The input data itself has no real significance in this thesis. The focus is on the algorithms which take sequences as input.

This thesis examines several aspects of the Four-Russians algorithm and its application. Gusfield’s method [1] for edit distances is used as a basis for the initial experiments, but the main focus is on local alignment.

The challenges of adapting the Four-Russians method to local alignment has been analyzed and solved, which includes deriving a general formula for encoding arbitrary weighted scoring systems. In practice local alignment with Four-Russians has a clear advantage over the naïve algorithm, even for slightly more complicated scoring systems. However, the gains of the method start to diminish once the encoding becomes more complex.

The preprocessing step in Four-Russians method has also been addressed, and a simple algorithm for computing the lookup tables very quickly was designed, analyzed and tested. Despite substantial speedups of the preprocessing step using this algorithm, it was shown to have little relevance when combined with the much slower Four-Russians step.

For completeness, speeding up local alignment backtracking using Hirschberg’s algorithm [6] in conjunction with Four-Russians has also been tested. The gains in backtracking are similar to the gain observed when computing the score of the alignment itself.
## Contents

1 Introduction  
   1.1 Structure of the thesis  
   1.2 Implementation  

2 Pairwise sequence alignment  
   2.1 The Wagner-Fischer algorithm  
   2.2 Memorization and time complexity  

3 Four-Russians  
   3.1 $t$-blocks  
   3.2 Encoding the $t$-blocks  
   3.3 Table size  
   3.4 Encoding considerations  

4 Data generation and testing  
   4.1 Generating pairs with similarity $x$  
   4.2 Implementation  
   4.3 Testing and hardware  

5 Global alignment experiments  
   5.1 Linear space consumption  
   5.2 LUT usage versus $t$ and $n$  
   5.3 Row- or column-major ordering  
   5.4 Naive vs Four-Russians  
   5.5 Naive vs Four-Russians 2  
   5.6 Encoding bits  

6 Local alignment and Four-Russians  
   6.1 Smith-Waterman algorithm  
   6.2 Problem 1 - Scoring systems and encoding  
   6.3 Problem 2 - Keeping track of the maximal index  
   6.4 Problem 3 - Handling the zero-case  

7 Generalizing neighbor distances  
   7.1 Moving on from unit cost edit distances  
   7.2 Arbitrary block sizes  
   7.3 Arbitrary scoring systems  

iv
## CONTENTS

7.4 Modifying Gusfield’s original proof ........................................... 35
7.5 Finding internal extreme values ................................................. 36

8 Speeding up LUT construction ...................................................... 39
8.1 Depth-first search (DFS) ............................................................. 39
8.2 Divide and Conquer ................................................................. 41
8.3 Combining the ideas ................................................................. 42
8.4 Local alignment ...................................................................... 45
8.5 Verifying correctness .............................................................. 45
8.6 Experiments ......................................................................... 46
8.7 Results ............................................................................... 46

9 Local alignment experiments ....................................................... 50
9.1 Four-Russians and t-block sizes ............................................... 51
9.2 t-block sizes with naive fallback ............................................. 53
9.3 Subtable usage .................................................................... 54
9.4 Input similarity effect on Four-Russians .................................... 55
9.5 Preprocessing vs Four-Russians .............................................. 56
9.6 Four-Russians and extended scoring system ............................. 58

10 Backtracking ........................................................................... 60
10.1 Naive backtracking ............................................................... 60
10.2 Hirschberg’s algorithm .......................................................... 61
10.3 Four-Russians and Hirschberg’s algorithm ............................... 64
10.4 Experiments ....................................................................... 66

11 Conclusion ............................................................................. 72
11.1 Future work ....................................................................... 73

Bibliography ............................................................................. 74
Chapter 1

Introduction

Pairwise sequence alignment is a problem that appears in many places in computer science, when comparing the similarity of two sequences of data. A general approach to solving sequence alignment is dynamic programming, upon which many algorithms are based. Four-Russians is a technique that can be used to speed up such algorithms. The Four-Russians technique was originally used to perform fast boolean matrix multiplication, and later modified by Dan Gusfield \[1\] to work on unit cost edit distances. Henceforth, Four-Russians will refer to Gusfield’s version.

This thesis will mainly be about exploring the possibility of using the technique to speed up local pairwise sequence alignment, in form of Smith-Waterman’s algorithm \[3\]. Local alignment and the SW algorithm are introduced later in the thesis. Related problems include finding a general way to encode weighted scoring systems for local alignment in Four-Russians, and performing backtracking utilizing the technique. These problems are addressed in the thesis. Additionally, an algorithm for quickly generating the lookup-tables needed by Four-Russians has been designed, and is analyzed and tested.

1.1 Structure of the thesis

The thesis is split up into a number of chapters. Chapter 2 introduces the concept of pairwise sequence alignment and the Wagner-Fischer \[2\] algorithm that can be used to solve the unit cost edit distance variant of the problem. A few key observations are summarized that become important later. The thesis starts out simply dealing with unit cost edit distances. In chapter 3 the Four-Russians algorithm and essential proof that makes it viable is described along with some of the terminology used throughout the thesis. Some practical considerations are also discussed. Chapter 4 and 5 deals with generating input test data and running experiment on the unit cost edit distance version of Four-Russians. Each experiment is given its own section with a small introduction and rationale for doing the experiment, followed by a discussion of the result.

The algorithm for local alignment is introduced in chapter 6, and the issues in adapting it to a Four-Russians variant are addressed and resolved. One of the key issues is finding a more general encoding than the one presented by Gusfield \[1\] which only works for unit costs. This is worked out in chapter 7 along with a solution to the other issues described in the chapter before.

The new algorithm for fast lookup-table construction is derived in chapter 8. Different variations of the algorithm are also tested here. In chapter 9 the experiments for local alignment are made. This chapter is structured the same way as chapter 5. The algorithm for fast lookup-table construction is also considered in conjunction with aligning the sequences. Finally, for completeness, backtracking local alignments using Four-Russians is discussed and tested in chapter 10 using Hirschberg’s algorithm \[6\].
1.2 Implementation

The code written for this thesis is available for download at http://cs.au.dk/~flababah/source.zip.

src/ contains the main implementation in C.

util/ contains some helper code in Python. The files prefixed with test. are used to generate plots. This folder is mainly included for completeness sake, since it also contains files which were used to test various things that might not make sense on their own.
Chapter 2

Pairwise sequence alignment

This chapter explains the base problem for this thesis: Pairwise sequence alignment and how it can be solved.

Given two sequences of data, $S_1[0..n]$ and $S_2[0..m]$, over a finite alphabet $\Sigma$, pairwise sequence alignment strives to find an optimal alignment between the pair such that the goal in some scoring system is minimized or maximized. This a problem that is often found in the field of bioinformatics where sequences of DNA or protein are compared. Two sequences are aligned by inserting a number of "gaps" in either sequence. Depending on the scoring system, the sequences

$$S_1 = \text{aaacagcatctccgg}$$
$$S_2 = \text{acgatcgtcctagca}$$

might yield a better score with gaps inserted, such that

$$S_1 = \text{aaacagcatc-tccg-g--}$$
$$S_2 = \text{a--c-g-atcgtcctagca}$$

The above alignment is an optimal unit cost edit script of two input sequences, in that it contains the minimum number of gaps and mismatches given the scoring system described in section 2.1. Several optimal alignments may exist for a given pair of sequences, which will all yield the same optimal score.

An insertion occurs in a sequence when a gap is inserted, typically in the first one. Note that we cannot actually remove elements from either sequence, so a deletion is the insertion of a gap in the opposing sequence.

2.1 The Wagner-Fischer algorithm

Finding an optimal alignment is a computationally intensive problem, since $n \cdot m$ distinct alignments of $S_1$ and $S_2$ exist. Wagner-Fischer’s algorithm \cite{2} can be used to solve the alignment problem.

$$T_{i,j} = \min \left\{ \begin{array}{ll} T_{i-1,j-1} + \text{score}(S_1[i-1], S_2[j-1]) & \text{if } i > 0 \text{ and } j > 0 \\ T_{i-1,j} + 1 & \text{if } i > 0 \\ T_{i,j-1} + 1 & \text{if } j > 0 \\ 0 & \text{if } i = 0 \text{ and } j = 0 \end{array} \right. \quad (2.1)$$

where
\[ \text{score}(\alpha, \beta) = \begin{cases} 
0 & \text{if } \alpha = \beta \\
1 & \text{else} 
\end{cases} \] (2.2)

The naive version of Wagner-Fischer is a recursive algorithm, given in equation 2.1. \( T_{i,j} \) is evaluated in order to find the alignment score of \( S_1[0..i] \) and \( S_2[0..j] \). \( T_{n,m} \) gives the score of the full pair of sequences, where \( |S_1| = n \) and \( |S_2| = m \). The function’s goal is to minimize the difference between the pair of sequences. It falls into the global alignment category since every element in each sequence is aligned. Henceforth, global alignment will refer to this algorithm and unit cost edit distance.

2.2 Memorization and time complexity

Simply evaluating the function above will result in exponential time usage. The crucial improvement, to make the algorithm feasible for larger sizes, is memorization. A table, size \((n+1) \times (m+1)\), can be used to store the intermediate values in the computation. It should come as no surprise that the run time for the algorithm becomes \( O(nm) \), or simply \( O(n^2) \) if the input pair have the same length, because each cell in the \( O(nm) \) table must be filled.

In practice all the bounds checks can be avoided by initializing the table such that

\[
T_{0,0} = 0 \\
T_{0,j} = j \text{ for } j = 1..m + 1 \\
T_{i,0} = i \text{ for } i = 1..n + 1
\] (2.3)

and only running the recursive function 2.1 on indices where both \( i \) and \( j \) are greater than 0.

2.2.1 Linear space consumption

The only advantage to the exponential version of the algorithm is the \( O(1) \) memory usage, if we of course disregard the call stack. Since each cell in the table is filled with values the space consumption becomes \( O(nm) \).

\[
\begin{array}{cccccccccccccccc}
\hline
& \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\hline
i = 0 & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline \\
\hline
i = 1 & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline \\
\hline
i = 2 & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline & \hline \\
\hline
\end{array}
\]

\[ \text{Figure 2.1: Filling the table in a row-wise manner} \]

The table needs to be filled in such a way that the left, upper, and upper-left neighbor have always been computed before a given cell. A simple, yet effective, scheme is to compute the table row by row, from left to right. If we are simply interested in the value of \( T_{n,m} \) alone, the other values inside the table only matters as long as there is an uncomputed neighbor that depend on those values. \( O(m) \) space usage can be achieved by only keeping row \( i \) and \( i - 1 \) in memory. Once
row $i$ is fully computed row $i - 1$ becomes irrelevant since no more cells depend on it. In fact, only one row and a temporary variable is needed if we memorize the current row and update in-place such that

$$
\langle T_{i,0}, T_{i,1}, ..., T_{i,m} \rangle = \begin{cases} i, \\
\min \left\{ \begin{array}{l}
p + \text{score}(X[i - 1], Y[0]) \\
R_1 + 1 \\
R_0 + 1 \\
\vdots \\
\min \left\{ \begin{array}{l}
p + \text{score}(X[i - 1], Y[j - 1]) \\
R_j + 1 \\
R_{j-1} + 1 \\
\end{array} \right. \\
\end{array} \right. 
\right) \tag{2.4}
$$

where $p = T_{i,j-1}$ before it is updated in step $j - 1$. The initial row is initialized using equation 2.3.
Chapter 3

Four-Russians

Four-Russians is a technique that can be used to speed up dynamic programming problems, like Wagner-Fischer from the previous chapter. The idea was originally used for boolean matrix multiplication, developed by four Russians where, as it turns out, only one was in fact Russian. However, the Four-Russians method has a better ring to it than One-Russian-and-three-other-blokes.

The idea was modified by Dan Gusfield in 1997 \[1\] to work on sequence alignment. More specifically, for computing the unit cost edit distance in a pair of sequences. This chapter summarizes his work and main proof, and will discuss some practical aspects of the algorithm and implementation. Also, ideas for fine-tuning the algorithm are discussed at the end of the chapter.

The original dynamic programming algorithm runs in $O(n^2)$ time. The Four-Russians \[1\] technique runs in $O\left(\frac{n^2}{\log^2(n)}\right)$ time. This is assuming $\log(n)$ is a number that fits into a word on the computer. For the remainder of the thesis this will be the case.

Since the input is given by a pair of sequences, it makes sense to distinguish their lengths. The execution time is then given by $O(nm/(\log(n) \log(m)))$. In practice the $\log(n)$ and $\log(m)$ component is substituted for some constant value, denoted $t$.

\begin{figure}[h]
\centering
\begin{tikzpicture}
\node at (0,0) {$S_1[0..n]$};
\node at (4,0) {$S_2[0..m]$};
\node at (2,2) {$A$};
\node at (3,2) {$B$};
\node at (4,2) {$B$};
\node at (5,2) {$B$};
\node at (2,1) {$C$};
\node at (3,1) {$C'$};
\node at (4,1) {$C'$};
\node at (5,1) {$C'$};
\node at (2,0) {$B'$};
\node at (3,0) {$B'$};
\node at (4,0) {$B'$};
\end{tikzpicture}
\caption{Four-Russians splits the dynamic table into a grid of blocks, which can be computed faster. One block’s output edges becomes its neighbors input edges.}
\end{figure}
3.1 \(t\)-blocks

Given a table of size \((n+1) \times (m+1)\), we wish to find the value of the lower-right corner after populating all the entries in the table using recursive function \(2.1\). Instead of filling the table one cell at a time, the Four-Russians method will instead split the table into a grid of smaller blocks, of size \((t+1) \times (t+1)\). These blocks are called \(t\)-blocks. See figure 3.2. If a table is \((n+1) \times (m+1)\), using \(t\)-blocks of size \(t = 3\) will result in the computation of a size \([n/3] \times [m/3]\) table, because the edges overlap. See figure 3.1. Since each \(t\)-block is essentially a small subtable in the original table, we can easily compute a \(t\)-block in \(O(t^2)\) time, using the same recurrence as we would on each individual cell. However, this does not improve the running time since the same amount work is done. The main idea is to precalculate information about these \(t\)-blocks in order to reduce the cost of each subtable computation, such that we do not have to spend \(O(t^2)\) time on each \(t\)-block. All the possible combinations of input to the \(t\)-blocks can be stored in a lookup-table, which will hereafter be shorted as \(LUT\).

A cell at \((i, j)\) is calculated by looking at its neighbors at \((i-1, j), (i, j-1)\) and \((i-1, j-1)\) and the symbols in the sequences at \(S_1[i-1] \) and \(S_2[j-1] \). The normal computation can be seen as computing blocks of size \(t = 1\). In the same way, an arbitrarily sized \(t\)-block is dependent on the same information, just an additional \(t - 1\) cells further down and to the right.

Consider figure 3.2. Cell \(A\) is the value at index \((i, j)\). Row \(B\) \((\text{B}_0, \text{B}_1, \ldots, \text{B}_t)\) depends on the values \((i-1,j+k)\) where \(k \in [0..t+1]\) and column \(C\) depends on cells \((i+k,j-1)\) where \(k \in [0..t+1]\). All the internal cells in the \(t\)-block (i.e. not \(A\), \(B\) or \(C\)) of course depend on their neighbors, but also on the subsequences \(S_1[i..i+t]\) and \(S_2[j..j+t]\). These subsequences used by the \(t\)-blocks will be denoted \(X\) and \(Y\), respectively.

Note that in Gusfield’s original article he denotes \(t\) as the width and height of the \(t\)-block including row \(B\) and column \(C\). Throughout this thesis \(t\) will denote the size of the \(t\)-block excluding \(B\) and \(C\). More specifically, we will say a block has size \(t_{in} \times t_m\), where \(t_n\) is the height of the \(t\)-block’s body and \(t_m\) its width, since the \(t\)-blocks does not necessarily have to be quadratic in shape. From here on, \(t_n\) and \(t_m\) will denote the \(t\)-block’s height and width, receptively excluding \(B\) and \(C\). For instance, a \(t\)-block with a body size of \(2 \times 3\) will be denoted \(t_{2\times3}\). Note that \(X\) and \(Y\) are called \(D\) and \(E\) in Gusfield’s original article, respectively.

The computation of the table is done by overlapping the \(t\)-blocks such that the output of a block becomes the input of the consecutive block, i.e. output \(C’\) in one block becomes input \(C\) in the next. See figure 3.1 and 3.2. Because the work performed in each direction is reduced by a factor \(t = \log(n)\), the Four-Russians method runs in \((n^2/\log^2(n))\) time, assuming that input and output of the \(t\)-blocks can be handled in \(O(1)\) time and that the lookup table has been created in advance.

3.2 Encoding the \(t\)-blocks

The input of the \(t\)-blocks should be encoded in such a way that a lookup table can be used. Precomputing all possible \(t\)-blocks is not feasible without some modification. By definition each cell in the table can have a value from 0 to \(n\). Encoding \(A\), \(B\) and \(C\) result in \((n+1)^{t_n+t_m}\) combinations. For the alphabet \(\Sigma\) of symbols in the input sequences, we have \(|\Sigma|^{t_n+t_m}\) combinations, resulting in a grand total of \((n+1)^{t_n+t_m}|\Sigma|^{t_n+t_m}\) distinct \(t\)-blocks. Each computation of a \(t\)-block takes \(O(t_{n}t_{m})\) time. Even if sufficient memory is available, this number clearly still adds up to a number way higher than what is required by the naive algorithm, which computes \((n+1)^2\) cells.

The main trick in the Four-Russians algorithm that makes it feasible in practice is the observation that a cell’s value only differ by a small amount from its neighbor’s. Thus, if \(A\) is
known we can simply encode $B$ and $C'$’s value using an offset of the adjacent cell. Gusfield’s proof is described below:

**Lemma 12.7.2** [1]. A cell’s value differ by at most 1 from any of its neighbors, that is $T_{i,j} - 1 \leq T_{i,j} \leq T_{i,j} - 1 + 1$.

Proof of upper bound $T_{i,j} \leq T_{i,j} - 1 + 1$. It follows from the definition in case of a gap that $T_{i,j} = T_{i,j} - 1 + 1$. Since the function minimizes, $T_{i,j}$ can get a lower value depending on the other neighbors. Thus $T_{i,j} \leq T_{i,j} - 1 + 1$.

Proof of lower bound $T_{i,j} - 1 - 1 \leq T_{i,j}$.

Left case: The path ends with a horizontal edge. Such an edge is a gap and $T_{i,j} = T_{i,j} - 1 + 1$. It is easy to see that $T_{i,j} - 1 - 1 \leq T_{i,j}$ also holds.

Right case: The path ends with a diagonal edge followed by $k - 1$ vertical ones. If $k = 1$ the path simply ends with a diagonal edge. Since there is a comparison and $k - 1$ potential gaps we get: $T_{i,k,j} - 1 + cmp + (k - 1) \leq T_{i,j}$. The observation used in upper bound can be modified:
3.3. **TABLE SIZE**

\[
T_{i,j} \leq T_{i-1,j} + 1
\]
\[
T_{i,j} \leq T_{i-k,j} + k \quad \text{for some } k \geq 0
\]
\[
T_{i,j-1} \leq T_{i-k,j-1} + k
\]
\[
T_{i,j-1} - 1 \leq T_{i-k,j-1} + k - 1
\]

The term in (3.1) can be substituted and we get
\[T_{i,j-1} - 1 + \text{cmp} \leq T_{i,j} \]. \text{cmp} can have outcome 1 or 0, where 0 gives the biggest jump in the previous equation, thus \[T_{i,j-1} - 1 \leq T_{i,j} \].

The two cases both satisfy the bound. The proof for lower- and upper bound also hold with row- and column coordinates interchanged.

**Example:** The edge \(AB_0B_1B_2 = [4,5,5,4]\) is encoded as \([1,0,-1]\). This encoding is used for \(B, B', C\) and \(C'\). These edges’ values depend on \(A\), but the encoding does not change when the value of \(A\) does. This means that \(A\) is not a relevant value at all, since both the encoding and recursive function works in a relative way. However, there is no trivial way of reducing the data size needed to encode the input subsequences, \(X\) and \(Y\).

Let \(I\) be the interval containing the distinct values a relative jump can have. \(I = |\{-1,0,1\}| = 3\) in case of unit costs. Using the encoding scheme, the size of the lookup table becomes more manageable at \(I_n + I_m |\Sigma| I_n + I_m = (|\Sigma|)^{I_n + I_m}\). For the remainder of the thesis \(|\Sigma|\) will be a constant 4. Since sequence alignment is often used in bioinformatics the input is usually DNA-sequences containing the four nucleotides. The symbols in \(\Sigma\) are \(A, C, G\) and \(T\).

3.2.1 **Block function**

The inputs to the \(t\)-block, and LUT, can be seen as a function:

\[F(Y,B,X,C) = B',C'\]

which takes the encoded parameters, \(Y, B, X, C\), required to compute the outputs, \(B'\) and \(C'\), of the \(t\)-block.

In Gusfield’s original work he makes the retrieval operation in the table a \(O(t)\) operation, taking into consideration the \(t\)-block sizes that do not fit into a CPU register. However, this thesis is mostly about the practical aspects of the algorithm. The \(t\)-values are always assumed to be able to fit into a register. The operation is thus \(O(1)\).

![Figure 3.3: Layout of the index in case of block-size \(t_2 \times 3\) and 2 bits used for each encoding and symbol in \(\Sigma\). The index spans 6 + 6 + 4 + 4 = 20 bits resulting in \(2^{20} = 1,048,576\) combinations.](image)

The encoding layout used throughout the thesis, and implementation, will be \(YBXC\). \(Y\) occupies the most significant bits of the index, and \(C\) the least. The choice of layout is quite important as we will see later. \(Y\) will refer to that specific part of the encoding while \(S_2[j..l]\) will refer the the \(j..l\) subsequence of the second input-sequence. The same is true for \(X\).

3.3 **Table size**

The size of the table depends on a several variables: The subsequences combinations of \(X\) and \(Y\), the combinations of all encodings of \(B\) and \(C\), and of course on the chosen \(t_n\) and \(t_m\) values.
We also need to consider how much memory should be used to store each entry in the lookup table. For unit cost edit distance \( I = 3 \), as seen in section 3.2, and \( |\Sigma| = 4 \). The number of different input encodings for arbitrary \( t \)-block sizes is \((I|\Sigma|)^{t_n + t_m}\). For different values of \( t \) the number of combinations is given in table 3.1.

<table>
<thead>
<tr>
<th>Block size</th>
<th>Used #entries</th>
<th>Real #entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{1\times1} )</td>
<td>144</td>
<td>256</td>
</tr>
<tr>
<td>( t_{1\times2} )</td>
<td>1,728</td>
<td>4,096</td>
</tr>
<tr>
<td>( t_{2\times2} )</td>
<td>20,736</td>
<td>65,536</td>
</tr>
<tr>
<td>( t_{2\times3} )</td>
<td>248,832</td>
<td>1,048,576</td>
</tr>
<tr>
<td>( t_{3\times3} )</td>
<td>2,985,984</td>
<td>16,777,216</td>
</tr>
<tr>
<td>( t_{3\times4} )</td>
<td>35,831,808</td>
<td>268,435,456</td>
</tr>
<tr>
<td>( t_{4\times4} )</td>
<td>429,981,696</td>
<td>4,294,967,296</td>
</tr>
</tbody>
</table>

Table 3.1: Number of lookup table entries required for edit distance with different values of \( t \). "Real #entries" is the real number if we encode the edit distance offset using two bits.

\( B' \) and \( C' \) have the same lengths as \( B \) and \( C \), respectively, and need to contain the same amount of information. \( I^{t_n t_m} \) combinations can occur in the output of the function. It is possible to store \( B' \) and \( C' \) in one or two bytes depending on the value of \( t \) and the encoding scheme used. In practice it is best to pad to the nearest byte instead of storing the entries as an array of packed bit vectors to keep the decoding overhead low. The implementation uses four bytes per entry, since more output data is needed for each \( t \)-block later in the thesis, and to keep it consistent across different sizes of \( t \).

The offset distance \( I = 3 \) must be encoded in such a way that the result of the block function can be retrieved as fast as possible. The most obvious way is to allocate two bits for each offset, because \( \lceil \log_2(3) \rceil = 2 \). This does inflate the actual size of the lookup-table as shown in table 3.1. Other ideas for encoding non-power-of-two values are discussed in the next section.

3.4 Encoding considerations

The fact that \( |\Sigma| = 4 \) has the nice property that four values can be encoded using two bits with no waste. Keeping different parts of the encoding in separate bits with no dependency on each other makes things a lot simpler, but there are of course cases where the data encoding does not fit perfectly.

Each time a value is wastefully encoded, the waste is given by \( \frac{\text{available values}}{\text{used values}} \). In case of unit cost encoding, three values are used of the four available in two bits, which is the minimum required by this scheme. When encoding a \( t_{3\times3} \)-block, \( \left( \frac{3}{2} \right)^3 = 5.619 \) times the optimal space is used.

3.4.1 Other idea

It is possible to encode the input in such a way that the lookup table has size equal to the theoretical number of combinations. In the usual binary encodings we use base-two, and bit-shifts and masks are used to extract individual parts of the encoding.

For arbitrary bases, the left- and right-shift can be substituted by multiplication and division by a power of that base, respectively. The least-significant bits can be masked using the modulus operator.
For unit cost edit distances we would encode the offset in base-three instead of the naive base-four approach. A digit in base-three, (0,1,2), would used to encode an \((-1,0,1)\) offset. Since both \(B\) and \(C\) uses these jumps they could be combined in a \((t_n + t_m)\)-digit base-three number. Extracting the low part of the number, \(x\), is a matter of computing \(x \mod 3^t\). The high part can be extracted by dividing by 3. Getting individual values in the base-three number is accomplished by using modulus and divide by some power of 3 instead of using bit-shifts. \(X\) and \(Y\) already fit perfectly, assuming \(|\Sigma| = 4\). \(XY\) would be stored as the least significant part of the index, and \(BC\) would be the most significant part. Extracting either of \(BC\) and \(XY\) is simply done by bit-shifting and masking. This approach would not waste any memory since the low part, \(XY\), does not waste any entries, and the high part, \(BC\), simply result in a table that is not a size power-of-two.

In practice there is a fine line between a theoretical improvement and the overhead it imposes in practice. Both the modulus and divide operations are extremely expensive on modern CPUs. Even though they are likely strength-reduced into faster operations by the compiler, since the values of \(t\) are known in advance, there is still a lot of overhead in this idea. If the overhead of handling the encoding is greater than the cost of computing a \(t\)-block naively there is no point in the first place.

Since \(B'\) is passed directly into \(B\) and the same for \(C\), we do not need to decode these values during the Four-Russians algorithm, except when propagating the value of \(A\). See section 3.4.3 \(B\) and \(C\), however, should ideally be independent bit vectors so an index can be built fast. The bits needed to encode either \(B\) and \(C\) using naive scheme and the idea is given in table 3.2.

<table>
<thead>
<tr>
<th>(t)</th>
<th>(\log_2 (4^t))</th>
<th>([\log_2 (3^t)])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3.2: Number of bits used to encode \(B\) and \(C\) using difference encoding schemes.

Notice how no space-saving is gained from encoding two \(t_3\)-rows as a single number \((t = 6)\), as opposed to two separate numbers. Realistically, for \(t \leq 4\), the size of the lookup table can be reduced by a factor four since it is possible to shave off one bit for \(B\) and \(C\). However, the actual number of used entries is still the same. Only the waste is reduced.

This was not implemented because all the encoding ranges used in local alignment are chosen to fit a power-of-two interval. Only the unit cost interval is wasteful, but it is not the focus of this thesis. Also, either \(X\) or \(Y\) must occupy the most significant bits of the index for best performance, as we will see in chapter 5.

### 3.4.2 Comparison encoding idea

Instead of feeding the necessary subsequences to the block function, we can feed it the comparison outcomes of said subsequences. See figure 3.4. This will require an extra lookup table which takes two subsequences and returns the outcomes for all comparison combinations. The advantage of having a separate table for comparisons is that an increase in alphabet size will not explode the size of the main lookup table. For example, a \(t_{2 \times 2}\)-block will normally use \(\log_2 (|\Sigma|^{2+2}) = 8\) bits to encode the subsequences, 2 for each symbol. For unit cost edit distance there are only two outcomes of a comparison, 0 or 1. Four comparisons are made in a \(t_{2 \times 2}\)-block, thus encoding the outcomes will only need 4 bits.

This idea works best for small sizes of \(t\), and when there are only two outcomes of a comparison. If more than two outcomes are possible, more than one bit is needed to encode the outcome.
The encoding scheme from section 3.4.1 could be used to great effect in that case. However, any scoring system with more than two outcomes quickly becomes worse than the normal single table approach, as seen in figure 3.5.

![Diagram](image)

**Figure 3.4:** Left: Normal approach which computes \( F(Y,B,X,C) \rightarrow B', C' \) in a single lookup. Right: The \( t_n \times t_m \) outcomes for comparison of subsequences \( S_1[i..i+t_n] \) and \( S_2[j..j+t_m] \) are found in LUT. The result, along with \( BC \), is used as index in the main lookup table, LUT, in order to find \( B'C' \).

![Table](image)

**Table 3.5:** Bits needed to encode the subsequence part of the encoding for different sizes for \( t \). Left table is the normal approach where each symbol takes 2 bits. Middle and right takes the comparison outcomes as input. See figure 3.4. The right table use the modulo encoding from section 3.4.1 in order to work with three comparison outcomes.

The problem is that we would likely want to use more than two comparison outcomes rather than increasing the alphabet size. The normal encoding is used in the implementation.

### 3.4.3 Edge-sum precomputation

Profiling shows that, apart from looking up the block-function in the lookup table, the biggest performance hit comes from calculating the lower-right value of a block by looking at the encoded edge. Consider figure 3.6. Naively the lower-right value is given by \( A + \text{sum}(C) + \text{sum}(B') = A + \text{sum}(B) + \text{sum}(C') \). However, since we can easily calculate any of the initial edges, \((0,j)\) or
3.4. ENCODING CONSIDERATIONS

Figure 3.6: The value of the cell at the overlap of $B'$ and $C'$ can be found by computing the sum of $B + C'$ or $C + B'$, added to the value of $A$. Walking both of $B$ and $C$ can be avoided by starting the walk at the $A$ of the computation at $(i,j - t)$ or $(i - t,j)$.

$(i,0)$, we only need to propagate a previous know value with $\text{sum}(B')$ or $\text{sum}(C')$. For instance, in a column-wise computation, in $(i,j)$, we always know value of:

$(i,j - t)$ This value can be looked up in the column buffer and the lower-right value of $(i,j)$ is $\text{buffer}[i] + \text{sum}(B')$.

$(j - t,i)$ This is the value prior to the current computation or one of the initial cells. This can simply be saved in a variable $\text{prev}$ such that the current lower-right value is $\text{prev}\text{+sum}(C')$.

However, decoding the edge and accumulating its sum, while not expensive, does add some overhead to the already light algorithm. Since number of combinations in such an edge is $I^t$ (where $t$ is either $t_n$ or $t_m$ depending on which edge is used) it makes sense to precompute a small table containing the sum of all possible edges. This gave roughly a 20% speedup in practice.

3.4.4 Input precomputation

The input sequences have to be split into a pair of size $\lceil n/t_n \rceil$ and $\lceil m/t_m \rceil$ encoded chunks. When computing in a column-wise manner, input chunk $S_2[j..j + t_m]$ can be computed before each new column and reused while iterating $i$ while $j$ stays fixed for that column. Thus, input sequence $S_2$ is only encoded once. However, each chunk in $S_1[i..i + t_n]$ is encoded for each column, which adds some overhead. Instead we can preprocess input sequence $S_1$ and save the encoded chunks in an array of size $\lceil n/t_n \rceil$. In practice this translated into a 30% speed boost.
Chapter 4

Data generation and testing

Some input data is needed for testing the algorithms and their implementations. Generating test data and how experiments were carried out is described in this chapter. A straightforward algorithm for generating sequence pairs of arbitrary similarity is also given.

The sequences were generated at random using the language’s built-in random number generator. Cryptographically secure randomness is not needed for testing purposes, so the pseudo-random number generator is sufficient.

Most of the input sequences were generated by picking uniformly at random from the alphabet \( \Sigma = \{A, C, G, T\} \). Real DNA sequences will likely not have a uniform distribution of symbols, but for the main testing input it is fine.

Other sequence pairs were generated in order to have a certain "similarity percentage". This becomes relevant when using the Four-Russians technique and local alignment, which is introduced in chapter 6. Generating a certain similarity in the input pair was achieved by modifying the weight of each symbol by picking uniformly from a string where each symbol can occur zero or more times. For instance, a similarity percentage of 50% of two sequences could be generated by picking from \( \text{ag} \) for the first sequence and from \( \text{gg} \) in the other.

Since alignment using dynamic programming is an \( O(nm) = O(|S_1| \cdot |S_2|) \) computation, the similarity percentage is given by the percentage of matches in all combinations of the two sequences. Formally it is:

\[
\sum_{e_1 \in S_1} \sum_{e_2 \in S_2} \frac{1}{n \cdot m} \text{ if } e_1 = e_2 \quad 0 \text{ else} = \frac{\sum_{\alpha \in \Sigma} (#\alpha \in S_1) \cdot (#\alpha \in S_2)}{n \cdot m}
\]

4.1 Generating pairs with similarity \( x \)

The theoretical run time does not depend on how well the two sequences match. In practice this might not be true for all algorithms, as we will see later on. It is possible to create a simple algorithm for generating a pair of base strings with a certain similarity. For instance the strings \( \text{accccggtt} \) and \( \text{aaaaaggtt} \) could be used as base for a percentage of 18%. Using these short base strings we can create arbitrarily long sequences by uniformly sampling from them. The sequences should converge towards the same similarity percentage as the base strings when they become sufficiently long.

The pseudo-code given in figure 4.1 can be used to generate a pair of base strings with a certain percentage. The algorithm keeps track of the number of each symbols in each of the base strings. Iteratively it will increase the count for a symbol in one of the strings by one, depending
\[ s_1 = \{ 1, 1, 1, 1 \} \] // Initialize symbol counts
\[ s_2 = \{ 1, 1, 1, 1 \} \] // I.e. base starts at "ACGT"
\[ \delta = \infty \quad i = 0 \]

\[ \text{while } \delta > C_{\text{error}} \text{ or } i < C_{\text{count}} \{ \]
\[ \quad \text{foreach str of } s_1, s_2 \{ \]
\[ \quad \quad \text{for } \alpha \text{ in randomized}(\Sigma) \{ \]
\[ \quad \quad \quad \text{str} [\alpha] += 1 \]
\[ \quad \quad \quad \delta = \text{abs} (\text{goal} - \text{similarity\_percentage}(s_1, s_2)) \]
\[ \quad \quad \quad \text{// Keep track of the best } \delta \text{ and } \alpha \]
\[ \quad \quad \quad \text{str} [\alpha] -= 1 \]
\[ \quad \quad \} \]
\[ \quad \quad \text{str}[\alpha_{\text{best}}] += 1 \]
\[ \quad \} \]
\[ \quad i += 1 \]
\[ \} \]

Figure 4.1: Pseudo-code for generating base strings with a certain similarity percentage.

on which symbol creates a new similarity which is closest to the goal. This step is looped until
the strings have a certain length, \( C_{\text{count}} \), or the similarity is sufficiently close to the goal, \( C_{\text{error}} \).

Initializing the counts to all ones does mean some extreme percentages close to 0\% or 100\% are not possible. However, when starting at all zeroes, the algorithm will be a bit boring in that it will sometimes only use two symbols to obtain the desired percentage. This is not a perfect approach, but it serves its purpose nicely.

The algorithm also only works when there are only two outcomes of a comparison, match or mismatch. For matrices with many different comparison outcomes, something else will have to be used. Fortunately, this is not needed in any of the experiments.

## 4.2 Implementation

A fast language is required since the main focus in this thesis is on the performance of the Four-Russians method. The choice naturally fell on the C programming language. C does have a few shortcomings. The lack of templates was the biggest obstacle. The implementation should be able to handle a lot of small variations, such as encoding intervals. It is of course possible to implement it in a dynamic way, such that any configuration can be chosen at run-time, but the resulting code will not be optimal. Ideally we want the compiler to generate separate code for each configuration since it will be able to optimize better when these variations are constant values. The solution was to define semi-constant values as macros and create a small Python framework such that each test could easily be compiled with the right defines enabled or set.

Furthermore, the compiler is not always able realize the intend of the code. For instance, the fixed-depth recursive construct, used in the implementation figure 4.2 (left) can, with advantage, be rewritten as the pseudo-code on the right.

The base case check is removed and the compiler might be able to store local variables in registers since the recursion is unrolled and the depth is constant. In practice this technique gave a substantial (factor 2-3) speedup for the algorithm in section 8.3.
function rec_base(...) {
    // Handle base case
}

function rec(int i, ...) {
    if (i == 0) {
        // Handle base case.
        return;
    }
    // Do recursive work.
    rec(i - 1, ...);
}

#macro MAKE_REC(name, call_f) {
    function name(...) {
        // Do recursive work.
        call_f(...);
    }
}

MAKE_REC(rec1, rec_base);
MAKE_REC(rec2, rec1);
MAKE_REC(rec3, rec2);
...

Figure 4.2: The left construct can be rewritten as the pseudo-code to the right if the recursive depth is constant and fairly shallow.

4.3 Testing and hardware

Experiments were run on Linux 3.18 (64 bit), on an Intel Core i5-M520 2.4 GHz (2.933 GHz turbo) processor with 8 GB of DDR3 RAM. Like most modern processors it has a hierarchy of cache-levels, listed from fastest to slowest:

- 32KB of 4-way associative L1-cache (Instructions)
- 32KB of 8-way associative L1-cache (Data)
- 256KB of 8-way associative L2-cache (Data)
- 3MB of 12-way associative L3-cache (Data)

C code was compiled using the Clang compiler version 3.5.1 with all common optimization flags turned on. Performance-insensitive utility code was implemented in Python 2.7. Individual algorithm invocations were run one to five times depending on the input length, and the best run was recorded.
Chapter 5

Global alignment experiments

So far we have looked at the theoretical aspects of pairwise sequence alignment in form of edit distance and the Four-Russians method that can be used to speed up the $O(nm)$ computation by a factor $\log(n)\log(m)$. The naive algorithm and Four-Russians have been implemented and are tested in this chapter. All pairs of sequences were generated at random with uniform distribution from $\Sigma$. Each experiment is given its own subsection. All plots with input length as the x-axis are plotted on a logarithmic scale.

5.1 Linear space consumption

Memorization is necessary to make the recursive function feasible in practice. Storing the entire table is not needed when only the edit distance is to be found. Using the idea of only storing a single row in memory should not affect the $O(n^2)$ time bound, nor the final score. The naive quadratic space algorithm is tested against its linear space variant, in order to verify the time bound.

![Figure 5.1: Verifying the $O(n^2)$ time bound for naive linear and quadratic space.](image)

(a) Logarithmic y-axis

(b) Execution time divided by $n^2$.
CHAPTER 5. GLOBAL ALIGNMENT EXPERIMENTS

Result

See figure 5.1. Both algorithms follow the same bound, as expected. Using linear space is about twice as fast. The right plot might look subquadratic in execution time, but both versions start to converge towards a straight line at bigger input, as they should. Notice that the plot for quadratic space stops when $n > 34,000$. This is due to `malloc` failing to allocate extremely big sizes. An input pair, both of size 34,000, requires $(n + 1)^2 \cdot \text{sizeof(int)} \approx 4.6\text{GB}$ of memory, while the linear version only needs $136\text{KB}$ when using a single row. This is, unsurprisingly, a factor $n$ space saving. The linear space variant will be used as the naive baseline when comparing other algorithms.

5.2 LUT usage versus $t$ and $n$

The Four-Russians algorithm works by precomputing all possible combinations of $YBXC$. The theoretical limitation of the algorithm is the space consumption and computation of the LUT. If the usage is significantly less than the entire table, it might make sense to only precompute the necessary parts lazily or try to compress the LUT.

Given a $t$-size and a varying input size, an extra array of booleans is created, the same size of the LUT. Each index is then marked in the boolean-array upon lookup. Once the algorithm has terminated, the number of marked entries is counted and divided by the total size to get the actual usage percentage.

Result

See figure 5.2. Even the extreme $t_{3\times4}$ size is 50% utilized at input size around 50,000. $t_{3\times4}$ is the largest practical size. Input sizes greater than 50,000 are going to be tested, so it seems unlikely that there is anything to gain from lazy computation. This would also require a different, slower way to lookup entries if we were to avoid allocating 50% of the table without knowing exactly which entries would be needed.

5.3 Row- or column-major ordering

The theoretical running time of the Four-Russians algorithm is $O(n^2 / \log^2(n))$. In practice the value of $\log(n)$ is picked to be some constant $t$. Of course, the $t$-blocks do not necessary have to be quadratic. Asymmetrical sizes will also work.

The hypothesis is that the row/column-wise choice of the table computation matters a lot on modern processors with caching. Consider the table encoding layout, $YBXC$. Using the default row-wise model, the value of $Y$ potentially changes for each $t$-block computed since we move the block along input $S_2$ in the inner loop. Since $Y$ is the most significant part of the table encoding, i.e. the most significant bits, the lookup pattern is likely scattered all over the entire table.

Conversely, when executing column-wisely, the value of $Y$ stay fixed for each column-computation. The lookup pattern is then confined to the much smaller area of $BXC$. The CPU might be able to utilize its cache better when the lookups are made in a smaller area for each column. Saying that the row/column orientation matters is a bit misleading, since it is the encoding layout that makes the difference. For instance column-wise computation and layout $YBXC$ is equivalent to row-wise orientation with layout $XCYB$ in terms of memory access pattern. For practical reasons, the layout is fixed in the implementation.
5.3. ROW- OR COLUMN-MAJOR ORDERING

Result

See figure 5.3. Column-wise computation is, predictably, a lot faster in most cases. Only in the configurations where \( t_n + t_m \leq 3 \) is row-wise slightly faster. This is likely because the entire table fits into the fastest cache, so the layout does not matter. For instance the \( t_2 \times 3 \)-table only occupies 1MB of memory with about a fourth of that memory being real table entries.

However, both the row and column oriented implementation are identical, apart from the orientation of course, so it is a bit odd for their performances to differ at all.

Both orientations seems to prefer flat block relative to their inner-loop direction, as illustrated in figure 5.4. For the column-major variant the size of the lookup area \( BXC \) compared to the entire \( YBXC \) space is inversely proportional to the size of \( Y \), that is \( t_m \). The point is that the size of \( BXC \) does not increase that much with \( t_m \), while the amount of computation done with a big \( t \) is improved. Looking at \( t_2 \times 4 \) and \( t_4 \times 2 \) is a good example of this.

Figure 5.2: The percentage of the generated entries in the LUT which were actually used during Four-Russians for a given input- and \( t \)-block size.
CHAPTER 5. GLOBAL ALIGNMENT EXPERIMENTS

Figure 5.3: Aligning two random sequences of length 50,000 using different $t$-block sizes. The difference in execution time for both row- and column-major orientation is also shown.

Figure 5.4: "Flat" blocks seem to work well. The flatness is perpendicular to the computation direction.
5.4. NAIVE VS FOUR-Russians

Performance counters

Profiling the execution using the `perf` command might give insight in this. The most interstering stats are listed below for the comparison of $t_{2 \times 4}$ and $t_{4 \times 2}$. It should be noted that the stats cover three runs, which is why the number of memory loads can exceed $\frac{50000 \times 50000}{2}$ for $t_{2 \times 4}$ and $t_{4 \times 2}$.

For $t_{2 \times 4}$-blocks

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-dcache-loads</td>
<td>5,663,916,528</td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>386,927,453</td>
</tr>
<tr>
<td>LLC-loads</td>
<td>39,775,219</td>
</tr>
<tr>
<td>LLC-load-misses</td>
<td>23,184,585</td>
</tr>
</tbody>
</table>

Both block sizes yield less than 0.04% branch-misses, so this is not the issue. Both variants tries to load from the L1 cache (the fastest level) roughly the same number of times. $t_{4 \times 2}$ does it slightly more. Likely because it simply runs for longer. What is interesting is that $t_{4 \times 2}$ misses almost three times as many lookups as $t_{2 \times 4}$. Unfortunately, it is difficult to get L2 statistics for this particular CPU, and verifying that it is indeed the right stats. However, we can see that 39 million load attempts, for $t_{2 \times 4}$ end up in the LLC (last level cache, L3). And 58% of those loads ends up having to read from main memory. For $t_{4 \times 2}$ 308 millions reads are being made to the memory, about 13 times more memory access than $t_{2 \times 4}$. This combined with the fact that $t_{2 \times 4}$ makes three times better use of the L1-cache is likely the reason for the big performance difference in practice.

Profiling the row-major variant on $t_{4 \times 2}$ tells almost the same story as column-major on $t_{4 \times 2}$ with 649 million LLC-loads and 79% miss-rate on those.

5.4 Naive vs Four-Russians

From figure 5.3 we can deduce that the column-major variant should be the default from now on, since it is almost always both theoretically and practically faster. It is also only interesting to look at the $t$-sizess where $t_n \leq t_m$. So far the Four-Russians implementation has only been tested on input size $n = 50,000$, but the speedup relative to the naive linear version might not be constant when $n$ is increased.

Result

See figure 5.5. There are quite a lot of things going on in this plot. It can be concluded that there is no best $t$-size for all input sizes of $n$. In general the bigger $t$-sizes begin to make sense when the input grows very large. The small $t$-blocks, where $t_n + t_m \leq 4$, tend to converge towards a constant speedup factor after a certain point. This might also be the case for the bigger $t$-sizes given extremely large input. In general $t_{2 \times 4}$ seems to be a good choice for edit distance computation. The results seem to be somewhat consistent with Espeholt’s findings [4].
CHAPTER 5. GLOBAL ALIGNMENT EXPERIMENTS

Figure 5.5: Testing Four-Russians unit cost edit distance with different $t$-block configurations using symmetric input size, $n = m$. No initialization overhead.

5.5 Naive vs Four-Russians 2

Looking at the result of the previous experiment, it seems the big $t$ sizes start to gain an advantage over the smaller blocks when $n$ grows to very large sizes. At $n = m = 1,000,000$ it looks like $t_{3 \times 3}$ and $t_{3 \times 4}$ are beginning to catch up. This reason for this is because $Y$ stays fixed for a column computation as mentioned before. However, for large $t$-blocks the columns are usually so short that the cache for this particular area of the LUT does not have time to warm up, since even $BXC$ is fairly large. $t_{3 \times 4}$ uses 1GB of memory. When $Y$ is fixed, the memory lookups are restricted to $BXC$ which, for every $Y$, is 4MB in size. Only 3 values of 4 possible ones are used in $B_0$, so the actual area is less than 3MB. The L3 cache of the tested CPU is able to store just that. The point is that the bigger $t$-blocks might be preferred when the columns become so long that repopulating the cache for almost every column is worth the overhead.

Unfortunately, the algorithm still runs in $O(nm/(\log(n) \log(m)))$ time, so computing a pair of sequences larger that a few millions starts taking a lot of time. This test will instead fix the number of columns, $m$, at 10,000, while increasing $n$ to large sizes. This will have the same effect as $n = m$, only we do not have to run the algorithm for years, and are likely to see the same performance characteristics.

$t_{1 \times 1}$ and $t_{1 \times 2}$ were omitted from the test since they reach their maximum potential at fairly small input sizes.
Result

See figure 5.6. The hypothesis turns out to be somewhat correct. The bigger $t$-sizes are indeed faster for very large input. However, the size of the L3 cache does not seem to influence the result very much. Looking at table 5.1 we can see cache miss statistics for different sizes of $n$, using the $t_{3 \times 4}$-block. Even though the number of L3 misses grow with $n$, this is somewhat irrelevant since the number of loads from L3 relative to the total number of lookups is, unexpectedly, very low. L2 statistics are not possible, but only a small fraction of the misses from L1 end up touching L3, which would mean the majority of lookup hit the L1 or L2 cache, not L3 or memory.

Since the effective size of $BXC|\Sigma| |\Sigma| \cdot |\Sigma| \cdot \text{sizeof(entry)} \approx 546\text{KB}$, is larger than the size of the L1 and L2 cache combined there must be some pattern or area within $BXC$ that is accessed more frequently than the rest. Otherwise, the number of L1 cache misses would be much higher, and performance would suffer.

<table>
<thead>
<tr>
<th>$n$</th>
<th>L1 misses</th>
<th>Approx L2 misses</th>
<th>L3 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>11.84%</td>
<td>42.04%</td>
<td>53.64%</td>
</tr>
<tr>
<td>1,000,000</td>
<td>3.61%</td>
<td>18.95%</td>
<td>55.36%</td>
</tr>
<tr>
<td>10,000,000</td>
<td>2.50%</td>
<td>6.46%</td>
<td>73.23%</td>
</tr>
</tbody>
</table>

Table 5.1: Cache misses for $t_{3 \times 4}$ at different sizes of $n$, $m = 50,000$. 
5.6 Encoding bits

As mentioned in section 3.3, there is some waste in the table when encoding the unit cost edit distance. In order to see how harmful this waste is, the implementation is also tested when three bits are used instead of the usual two. The number of actual table indices stay the same.

Result

![Graph comparing 2-bit and 3-bit offset encoding]

**Figure 5.7:** Comparing the performance of Four-Russians using 2 and 3 bits per offset encoding. \( n = 100,000 \).

See figure 5.7. The loosely packed version does get quite a penalty when \( t \) becomes large. For \( t_{2\times4} \) using three bits will result in a table that is \((2^3|\Sigma|)^{2+4}/(2^2|\Sigma|)^{2+4} = 64\) times bigger, which result in roughly half the performance. Small sizes of \( t \) are not as vulnerable because of the smaller \( t \) and because they are more likely to fit into the cache no matter if the encoding bits are two or three.

It is definitely important to pack the used entries as tightly as possible. Profiling the \( t_{2\times4} \) case with 2 and 3 bits show that 3 bits creates 41\% more L1-misses and ends up reading from main memory 3.4 times as much. None of the variants miss the L1 cache that much (6.78\% versus 9.59\%), which just goes to show how expensive those misses that end up reading from main memory are.
Chapter 6

Local alignment and Four-Russians

We have seen how the Four-Russians technique can be used to speed up global alignment problems. However, the main focus of this thesis is local alignment. This chapter is about adapting the Four-Russians method to work with local alignment. The different obstacles of the chosen approach are addressed here.

<table>
<thead>
<tr>
<th></th>
<th>c</th>
<th>c</th>
<th>g</th>
<th>g</th>
<th>a</th>
<th>a</th>
<th>a</th>
<th>c</th>
<th>g</th>
<th>t</th>
<th>c</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>g</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 6.1: Local alignment looks for similarity in subsequences. An example of a scoring system with match=1, mismatch=-1 and gapcost=-1 is shown. Two optimal alignments are highlighted, although more exist.

Global alignment is mostly used when the input is already similar and has roughly the same length. Pairwise Local alignment is about finding regions of maximum similarity in input that might be dissimilar for the most part. Many local alignment algorithms exist that can solve this particular problem. This thesis will focus on the Smith-Waterman [3] algorithm from 1981. BLAST [5] is a very popular algorithm in the real world, which is generally faster than the dynamic programming approach of SW. However, whereas BLAST can produce several suboptimal alignments, SW will always find an optimal one, but only one. SW can be modified to find all optimal alignments, but it will most likely violate its $O(n^2)$ time bound. The simplicity of SW and its similarity to the original edit distance problem makes it an obvious choice for adapting to a Four-Russians variant.
6.1 Smith-Waterman algorithm

Like the Wagner-Fischer algorithm for edit distance, SW is also based on dynamic programming. At first glance the recursion almost look the same:

\[
T_{i,j} = \max \begin{cases} 
T_{i-1,j-1} + \text{score}(S_1[i-1], S_2[j-1]) & \text{if } i > 0 \text{ and } j > 0 \\
T_{i-1,j} + \text{gap} & \text{if } i > 0 \\
T_{i,j-1} + \text{gap} & \text{if } j > 0 \\
0 & \text{otherwise}
\end{cases}
\]  

(6.1)

Several things are different compared to edit distance:

1. The score-function is more loosely defined here. It is usually a matrix, $|\Sigma| \times |\Sigma|$, which contains a score for each combination of input symbols. For the majority of testing the scoring function will just have two different values. One for match and one for mismatch. A constant gap penalty is used such that each gap has a fixed penalty value.

2. The point of local alignment is to not align both sequences to their full length. The zero-case is used to start at new alignment. The best alignment ends at the highest value in the table. The algorithm needs to keep track of these coordinates.

3. The objective to do maximization instead of minimization, and that the zero-case is now used on all cells, not just the upper-left index. This effectively mean that no value in the table will ever get below zero and that the alignment is ”restarted” if this happens.

In case of using the Four-Russians method to compute local alignment, the main idea is still the same as in global alignment. However, the three problem points are addressed in the following sections.

6.2 Problem 1 - Scoring systems and encoding

Unit cost edit distance has an encoding interval of $\{-1, 0, 1\}$, which Gusfield has proved. It is easy to see that many other scoring systems will not work with this encoding scheme. For instance, the encoding instantly breaks if we pick $\text{gap} = -2$. Since the gap penalty is constant the layout of the Four-Russians function does not need to be changed for this step. However, a new encoding scheme must be found since the scoring system for unit costs is not going to be used for local alignment. A more general encoding formular, for weighted edit distance, is derived in chapter 7. Ideas for encoding the encoding interval were discussed in section 3.4.

6.3 Problem 2 - Keeping track of the maximal index

Of course if we only need to know the score of the optimal alignment we just need to keep track of the highest value in the table. The coordinates of the best index are needed for backtracking. For plain SW this is trivial to keep track of while the algorithm runs. The original Four-Russians function can be modified to work with this change.

Each $t$-block should also in some way encode the maximal value within. It seems the straightforward modification is to store the maximal cell of the $t$-block as a relative value depending on $A$. $A$ is already available in the computation since the output of the $t$-block depends on it. An extra field needs to be added to the encoded output, such that the block function becomes:

\[
F(Y,B,X,C) = B', C', \max
\]  

(6.2)
Throughout the algorithm a new best $t$-block is found when $A + t_{\text{block}}.\max > \text{current}\_\text{best}$. At the end of the computation we know in which $t$-block the maximal cell resides. However, the precise cell index still has to be found. This is most easily achieved by storing the necessary information needed to recompute the final $t$-block naively. Since we are not interested in the exact best location during the computation we can defer this step to the end and simply save the current coordinates and $A$-value resulting in the new best $t$-block, and reconstruct the table for that block in the end. Storing the information during the computation takes $O(1)$ time and reconstructing the $t$-block and scanning for the best subvalue takes $O(t_n t_m)$ time.

Things get slightly more tricky when the input size is not evenly divisible by the block size. The problem presents itself when a new best $t$-block is found, but crosses the border of the table such that $i + t_n > n/t_n$ or $j + t_m > m/t_m$. The partially invalid input is normally not a problem because it cannot affect the real values inside the table.

Even if the exact location of the maximal value in the $t$-block was known, we would not be able to discard it. If the maximal cell is out of bounds there still might be an equally valued cell in the $t$-block that is within bounds. See figure 6.4. Storing the location of all maximal cells in a $t$-blocks is not feasible space-wise.

**Figure 6.2:** Prefixing the sequences with matching symbols might produce an incorrect result for local alignment.

**Figure 6.3:** Padding would need a new symbol $\alpha \in \Sigma$ to work correctly.

**Padding:** One solution to the problem is to prefix the input sequences with padding so they are evenly divisible by the $t$-blocks. In global alignment this is simply achieved by prefixing both sequences with the same symbols, such that the first part of the alignment matches. Since local alignment works on smaller regions, this approach might produce a wrong alignment if one is found in the padding. See figure 6.2 for an example where the actual pair is mainly dissimilar.

Instead a new sentinel symbol $\alpha$ would have to be added to $\Sigma$ if at least one axis need padding. See figure 6.3. This padding can only be verified the be correct if $\text{score}(\alpha, \beta) \leq 0 \text{ for } \forall \beta \in \Sigma$. Also $\text{score}(\alpha, \alpha)$ must be $\leq 0$. The zero-case of SW would ensure that the actual initial edges are all 0.

$A$, $C$, $G$ and $T$ are already encoded in two bits without loss. A new symbols would increase the number of needed bits to three. For at $t_{3x3}$-block this would make the lookup table $2^{3+3} = 64$ times bigger. An extreme solution to what is essentially an edge case. The idea from section 3.4.2 could also be used.
**Sliding:** The idea of sliding the final row- and column-blocks back into bounds is briefly discussed in section 10.3.

**Naively:** When a $t$-block is partially out of bounds we can fall back to computing it naively. One trick is to first compute the block normally using the lookup table on the partially invalid input. If the result does not yield a better maximum anywhere in the $t$-block, there is no need to invoke the naive algorithm. Otherwise, we run the naive and check if there indeed is a new maximum within bounds. This is idea chosen for the implementation.

![Figure 6.4: Determining the maximum value of a $t$-block across a boundary requires some care, since several maximal cells might exist.](image)

### 6.3.1 Maximal index consistency

Both the implementation of the naive and the Four-Russians version will record the first occurrence of the maximum value. Since the maximum value can occur several times throughout the computation these implementations might not yield the same position. That is unless we modify the naive version to access the table in $t$-sized blocks at a time, or make the Four-Russians only use flat $t$-blocks, either $t_n$ or $t_m = 1$, in order to emulate the pattern used by the naive algorithm. As long as they produce the same score, this is OK. The maximal coordinates cannot be verified correctly without backtracking since several optimal alignments may exist.

### 6.4 Problem 3 - Handling the zero-case

For edit distance all parts of the recursion works relative to $A$. The SW algorithm, however, has the absolute floor value of zero which cannot be encoded in a relative way. We can of course extend the block function such that any value of $A$ is encoded in the input as well:

$$F(A, Y, B, X, C) = B', C', max$$  \hspace{1cm} (6.3)

But again, we are looking at $O(n + m)$ different values of $A$, which is unfeasible as concluded in the beginning of the thesis.

One observation is that the output stops changing when the value of $A$ becomes sufficiently large. This is because the zero-case in the recurrence is never used if the input of the $t$-block never drop low enough to cause any cells inside the $t$-block to use it. The idea is then to create a
lookup table for each value of $A$ below this limit and one table which handles all lookups where $A \geq \text{limit}$ where the zero-case cannot have an effect on the outcome. The lowest cell in a $t$-block, $V_{\text{min}}$, is dependent on the scoring system and $t$-sizes.

If $V_{\text{min}}$ is known, the number of distinct LUTs that needs to be created is:

$$\#\text{LUTS} = -V_{\text{min}} + 1 \quad (6.4)$$

For instance $V_{\text{min}} = -3$ means that $A = 3$ is the smallest value of $A$ where the zero-case is never touched. Thus a table for $A \geq 3$ will have to be created along with table for $A = \{2, 1, 0\}$, which gives $-V_{\text{min}} + 1 = 4$ tables in total. Even if all the parameters in the scoring system are positive $V_{\text{min}}$ will never get above 0, because $A$ relative to itself is 0.

If the input in a lookup has $A \geq 3$, then the zero-case in the recursive function can clearly not affect the output of the block, since the lowest possible value will be 0 without considering the zero-case. However, when $A$ is 2, 1 or 0 the output might be affected, and a separate table or some other idea will have to be used to ensure a correct result.

Along with figuring out an encoding $I$ for a particular scoring system, the only thing that remains is finding $V_{\text{min}}$ for a given set of parameters. Both of which is done in the next chapter.
Chapter 7

Generalizing neighbor distances

The original proof for the unit cost encoding given by Gusfield is nice, but it does not immediately help us when it comes to other scoring systems. For unit costs we know the tightest interval needed to represent neighbor offset distances, but this interval $I$ will obviously not work for all scoring systems. This chapter is about finding a way to encode the jumps in value between cells for local alignment and arbitrary scoring systems. The original proof is then modified to verify the solution. We must also find a way of knowing $V_{min}$, the lowest relative value inside any $t$-block, in order for local alignment and Four-Russians to work using the ideas presented in chapter 6.

7.1 Moving on from unit cost edit distances

The idea that we are interested in the distance interval for any neighbor to a given cell is needlessly general in this context. Only the $B$, $C$, $B'$ and $C'$ blocks are encoded using this scheme, and they are only dependent on the neighbor above or to the left, never diagonally. This observation makes things a bit simpler, but it will likely also give a tighter encoding interval, $I$.

For this chapter a maximizing recursive function is considered. The zero-case in the local alignment function is ignored for now, which has the same effect as assuming cell $A$ to be a high value. $A \geq -V_{min}$ to be specific.

Both the comparison outcomes, a value in the interval $[cmp_{min}..cmp_{max}]$, and the gap-cost are added to neighboring cell values. Thus, gap should take a negative value for it to be a penalty.

$$D_{i,j} = \max \begin{cases} D_{i-1,j-1} + [cmp_{min}..cmp_{max}] \\ D_{i-1,j} + gap \\ D_{i,j-1} + gap \end{cases}$$

(7.1)

Usually a match has a certain value, likewise does mismatch. Of course, scoring systems exist where the comparison has a weighted result, which is not necessarily one of two values. For this reason two new variables, $cmp_{min}$ and $cmp_{max}$, are introduced which denote the lowest score of a comparison and the highest, respectively. In the simple case where there is a match and a mismatch, $match = cmp_{max}$ and $mismatch = cmp_{min}$.

**Definition 7.1.1.** An encoding interval, $I = [I_{min}..I_{max}]$, can be used in Four-Russians if it, for all possible inputs to a $t$-block, can encode both the input and output of the $t$-block, and the encoding covers the straight neighbor jumps created by initializing the initial row/column. I.e. $gap=-2$ gives initial interval $I = [-2..-2]$ which should be covered by the encoding.
7.2. ARBITRARY BLOCK SIZES

Proof of validity. The first row and column will be initialized to 0, gap, 2·gap, 3·gap,... This can be encoded if the definition is satisfied. After the initial rows and columns, the next t-blocks depend on the output of the previous ones. Since the encoding covers both the input and output, this will work for any size table.

The problem is that we do not fully know the encoding needed for representing a given scoring system. We could of course set the interval to that of a 32 bit integer, and be reasonably sure it would work. However, this is unfeasible in practice. Is it possible to work out the smallest needed interval, for a given scoring system, using a brute force algorithm described below:

1. Chose a t-block size that we want an encoding for.
2. Set the initial interval to \( I = [g\text{ap}..g\text{ap}] \), such that the interval covers only the gaps in the initial edge.
3. Iterate all possible inputs to the t-block, where \( B \) and \( C \) cell values are chosen according to the current interval \( I \).
4. Calculate the cells in the t-block for the given input. If in any cell calculation the jump from the left or upper cell to the current cell has a value outside the current interval, update the interval to contain this jump and start over at step 3.

The algorithm starts at the base assumption, that the interval only considers the gaps in the initial row and column. Each time a jump violates this assumption, the interval is updated to accommodate for this type of jump. If the algorithm finishes, we have a valid minimal encoding interval.

Instead of using an interval to represent the possible jump values, one could use a set to store each individual jump. For instance, using a set to calculate the encoding, the algorithm computes that the scoring system \( \text{match} = 5, \text{mismatch} = 2, \text{gap} = -2 \) only needs the values \((-2, 1, 4, 7)\) to represent the jump offsets. This might seem like a big deal since these offsets can be encoded using only two bits, when the entire interval would need \( \lceil \log_2 (7 - (-2)) \rceil = 4 \) bits. However, the number of different values quickly becomes the same as the interval’s when we consider scoring systems with more than two comparison outcomes or match/mismatch values closer to each other. This also will not work when we introduce the zero-case in the recursive function, since it could result in a value which is not a \(-2, 1, 4, \) or \(7\) offset from its neighbor, due to a cut-off by the zero-case.

7.2 Arbitrary block sizes

The above approach is slow and only guarantees that the encoding works for a chosen t-block size. But...

Lemma 7.2.1. Consider the algorithm in section 7.1. The encoding \( I \) generated, for a given scoring system, is the same for a \( t_{1 \times 1} \)-block as it is for a \( t_{n \times m} \)-block where \( n \geq 1 \) and \( m \geq 1 \).

Proof. Any internal calculation in the \( t_{n \times m} \)-block can be substituted by a \( t_{1 \times 1} \)-block calculation. Any jump that leads up to a certain subcalculation will have to be supported by the encoding interval, since the algorithm is restarted with any such jump included in the search space. No situation inside the big t-block will ever arise that the \( t_{1 \times 1} \)-block will also not see in form of the direct input.

This makes determining the correct interval for a given t-block size much faster. It is still a brute force approach, so something more elegant might be possible.
7.2.1 Inside the $t_1 \times t_1$-block

Instead of iterating all the possible inputs we can try to be a bit more clever by analyzing the cases which result in the extreme points of the interval. Consider the green cell in figure 7.1.

![Figure 7.1: Extreme offset values in the $t_1 \times t_1$-block. The current interval is given by $I = [I_{\text{min}}..I_{\text{max}}]$. The input edges $I_{\text{min}}$ and $I_{\text{max}}$ recursively depend on the jump distance to the green cell from top or left.](image)

First we want to find the value of $I_{\text{max}}$. This is defined by the biggest increment in value in a jump from one cell to the cell down or to the right. We do not consider cell A (labeled 0) a neighbor since it is positioned diagonally to the green cell. An assumption is made that only the extreme values in the comparison matter when extending the encoding, therefore we only consider $\text{cmp}_{\text{min}}$ and $\text{cmp}_{\text{max}}$. $I_{\text{max}}$ is the jump from the smallest cell value to the biggest. To be safe we consider the three cases:

1. Both $I_{\text{min}}$ and $I_{\text{max}}$ are present in $B$ or $C$. We only consider $\text{cmp}_{\text{max}}$ for $I_{\text{max}}$ since it will give the biggest jump destination. Also we can jump from both $B$ or $C$ which, due to the encoding, can span the interval $[I_{\text{min}}..I_{\text{max}}]$. We want that value to be the smallest.

\[
I_{\text{max}} = \max \left\{ I_{\text{max}} + \text{gap} \right\} \left(7.2\right)
\]

2. Only $I_{\text{min}}$ is present in $B$ and $C$. That means it is only possible to jump from $I_{\text{min}}$.

\[
I_{\text{max}} = \max \left\{ I_{\text{min}} + \text{gap} \right\} - I_{\text{min}} \left(7.3\right)
\]

3. Finally when only $I_{\text{max}}$ is the neighbor. Like above, it becomes the only jump source.

\[
I_{\text{max}} = \max \left\{ I_{\text{max}} + \text{gap} \right\} - I_{\text{max}} \left(7.4\right)
\]

In equation 7.2 $I_{\text{min}} + \text{gap}$ can be removed from the first term since it is always the case that $I_{\text{min}} \leq I_{\text{max}}$. Likewise, $I_{\text{max}}$ can be removed from the second term. Combining the equations is simple. We are interested in the maximum value:
7.2. ARBITRARY BLOCK SIZES

\[ I_{\text{max}} = \max \begin{cases} I_{\text{max}} + \text{gap} - I_{\text{min}} \\ I_{\text{max}} + \text{gap} - I_{\text{max}} \\ I_{\text{min}} + \text{gap} - I_{\text{min}} \\ \text{cmp}_{\text{max}} - I_{\text{min}} \\ \text{cmp}_{\text{max}} - I_{\text{max}} \end{cases} \]

\[ = \max \begin{cases} I_{\text{max}} + \text{gap} - I_{\text{min}} \\ \text{gap} \\ \text{cmp}_{\text{max}} - I_{\text{min}} \\ \text{cmp}_{\text{max}} - I_{\text{max}} \end{cases} \]

\[ = \max \begin{cases} I_{\text{max}} + \text{gap} - I_{\text{min}} \\ \text{cmp}_{\text{max}} - I_{\text{min}} \end{cases} \]  \hspace{1cm} (7.5)

\( I_{\text{min}} \leq I_{\text{max}} \) helps reduce the expression quite a bit. Moving on to finding \( I_{\text{min}} \), the main idea is the same, the only difference being that the first term should be minimized and the second maximized to give the smallest possible jump. The three cases are almost the same:

\[ I_{\text{min}} = \max \begin{cases} I_{\text{max}} + \text{gap} \\ I_{\text{min}} + \text{gap} - \max \begin{cases} \text{cmp}_{\text{min}} - I_{\text{min}} \\ \text{cmp}_{\text{min}} - I_{\text{max}} \end{cases} \end{cases} \]

\[ = \max \begin{cases} I_{\text{max}} + \text{gap} \\ \text{cmp}_{\text{min}} - I_{\text{min}} \end{cases} \]  \hspace{1cm} (7.6)

\[ I_{\text{min}} = \max \begin{cases} I_{\text{min}} + \text{gap} - I_{\text{min}} \end{cases} \]  \hspace{1cm} (7.7)

\[ I_{\text{min}} = \max \begin{cases} I_{\text{max}} + \text{gap} - I_{\text{max}} \end{cases} \]  \hspace{1cm} (7.8)

The above three functions can also be combined and reduced. Since it is now \( I_{\text{min}} \), the minimum case of above is wanted, although the recursive function is still maximizing.

\[ I_{\text{min}} = \min \begin{cases} \max \begin{cases} I_{\text{min}} + \text{gap} - I_{\text{min}} \\ \text{cmp}_{\text{min}} - I_{\text{min}} \\ I_{\text{max}} + \text{gap} - I_{\text{max}} \\ \text{cmp}_{\text{min}} - I_{\text{max}} \end{cases} \end{cases} \]

\[ = \max \begin{cases} \text{gap} \end{cases} \]

\[ = \max \begin{cases} \text{gap} \end{cases} \]

\[ = \max \begin{cases} I_{\text{max}} + \text{gap} - I_{\text{max}} \end{cases} \]  \hspace{1cm} (7.9)

Since the function 7.5 and 7.9 are co-dependent, an encoding for a given scoring system can be found by iteratively evaluating the functions until a fix point is reached. \( I_{\text{min}} \) and \( I_{\text{max}} \) are initially set to the gap score since this is the only initial information available that any encoding should be able to handle.
7.3 Arbitrary scoring systems

It is not clear from looking at function 7.5 and 7.9 how long they will take before they arrive at a fix point, or if they ever will. Looking at different combinations of match, mismatch and gap it can be observed that the minimum of match and mismatch never seem to affect the result. Table 7.1 shows a small sample output.

<table>
<thead>
<tr>
<th>match</th>
<th>mismatch</th>
<th>gap</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>-2</td>
<td>-2.5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>-1</td>
<td>-1.4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0.3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>-2</td>
<td>-2.5</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>-1</td>
<td>-1.4</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>0</td>
<td>0.3</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
<td>2</td>
<td>2.2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-2</td>
<td>-2.5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>-1</td>
<td>-1.4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2.2</td>
</tr>
</tbody>
</table>

Table 7.1: Running the bruteforce algorithm to obtain $I$. $\text{cmp}_{\text{min}}$ does not seem to affect the produced encoding interval.

If we make the assumption that the encoding does not depend on the minimum of match and mismatch, i.e. $\text{cmp}_{\text{min}}$, function 7.9 and 7.5 can be simplified even further.

$$I_{\text{min}} = \max \left\{ I_{\text{max}} \frac{\text{gap} + \text{gap}}{\text{cmp}_{\text{max}}} - I_{\text{max}}, \text{gap} \right\} = \frac{(I_{\text{max}} + \text{gap}) - I_{\text{max}}}{\text{gap}}$$

(7.10)

$$I_{\text{max}} = \max \left\{ I_{\text{max}} \frac{\text{gap} + \text{gap}}{\text{cmp}_{\text{max}}} - I_{\text{min}}, \text{gap} \right\} = \max \left\{ \text{gap} \right\}$$

(7.11)

Now the functions are also independent of each other and can be computed in $O(1)$ time. Notice how $I_{\text{max}}$ becomes $\text{gap}$ in 7.11. This is due to the initial constraint that the encoding should be able to handle gaps. The correctness of the encoding interval given by 7.10 and 7.11 have been verified by running the naive encoding algorithm in section 7.1 on a $t_2 \times 2$-block with
7.4. MODIFYING GUSFIELD’S ORIGINAL PROOF

all combinations of the values $[-10..10]$ for \textit{match}, \textit{mismatch} and \textit{gap}, and on a $1 \times 1$-block with all combinations of the parameters in the range $[-50..50]$. The result of that algorithm has been compared to the above functions and is correct for that range.

The encoding interval for a maximizing recursive function is thus given by:

$$I = \left[ \text{gap} \ldots \max \left\{ \frac{\text{gap}}{\text{\textit{cmp}_{\text{max}}} - \text{gap}} \right\} \right]$$  \hfill (7.12)

7.3.1 Examples

For scoring system \textit{match} = 1, \textit{mismatch} = -1 and \textit{gap} = -1 we get the following encoding:

$$I = [-1..\max(-1, 1 - (-1))] = [-1..2]$$ \hfill (7.13)

These equations hold for a maximizing recursive function. We can also find the encoding for unit cost edit distance, which is minimizing, by negating the parameters. \textit{match} = 0, \textit{mismatch} = -1 and \textit{gap} = -1 gives:

$$I = [-1..\max(-1, 0 - (-1))] = [-1..1]$$ \hfill (7.14)

7.4 Modifying Gusfield’s original proof

As an alternate way of deriving the encoding, Gusfield’s encoding proof can be extended to consider arbitrary comparison- and gap costs with a maximizing function. The modification is based on the proof in section 3.2 and the approach is basically the same.

Gusfield’s original proof mentions that $D_{i,j} \leq D_{i-1,j-1} + 1$. In a maximizing function this would be $D_{i,j} \geq D_{i-1,j-1} + \text{\textit{cmp}}$. We don’t have to reason about this part, since it was concluded earlier that diagonal jump intervals are not relevant in Four-Russians.

As before, the lower- and upper-bound are given by:

$$D_{i,j-1} + I_{\text{min}} \leq D_{i,j} \leq D_{i,j-1} + I_{\text{max}}$$ \hfill (7.15)

But unlike before, we are now interested in deducing the values of $I_{\text{min}}$ and $I_{\text{max}}$ rather than proving the statement.

\textbf{Part 1. Lower-bound:} $D_{i,j-1} + I_{\text{min}} \leq D_{i,j}$

It follows from the function definition that $D_{i,j-1} + \text{gap} = D_{i,j} \implies D_{i,j-1} + \text{gap} \leq D_{i,j}$, since this is a maximizing function and it can never be the case that $D_{i,j-1} + \text{gap} > D_{i,j}$. Thus, $I_{\text{min}} = \text{gap}$.

\textbf{Part 2. Upper-bound:} $D_{i,j} \leq D_{i,j-1} + I_{\text{max}}$

\textbf{Horizontal end:} Path $D_{0,0} \rightarrow D_{i,j-1} \rightarrow D_{i,j}$. Upper-bound for that case is given by the gap cost:

$$D_{i,j} = D_{i,j-1} + \text{gap}$$ \hfill (7.16)

\textbf{Diagonal end:} Path $D_{0,0} \rightarrow D_{i-k,j-1} \rightarrow D_{i-k+1,j} \rightarrow D_{i,j}$ ends with a diagonal or vertical edge. If $k = 1$ then it is diagonal; vertical otherwise. For the diagonal step we are interested in $\text{\textit{cmp}_{\text{max}}}$ in order to get the largest jump. The expression for the path becomes:

$$D_{i,j} \leq D_{i-k,j-1} + (k - 1) \cdot \text{gap} + \text{\textit{cmp}_{\text{max}}}$$ \hfill (7.17)
We can generalize the expression from part 1 to \( D_{i,j} \geq D_{i-k,j} + k \cdot \text{gap} \) for any \( k \). Of course, this also holds when we subtract one from \( j \) to get \( D_{i,j-1} \geq D_{i-k,j-1} + k \cdot \text{gap} \). When \( \text{gap} \) is subtracted from each side, we get \( D_{i,j-1} - \text{gap} \geq D_{i-k,j-1} + (k-1) \cdot \text{gap} \). This first term can be substituted in the above equation because it is at least as great as the second term and we are looking for an upper-bound. We thus get:

\[
D_{i,j} \leq D_{i,j-1} - \text{gap} + \text{cmp}_{\text{max}}
\]  

(7.18)

In the original proof the two cases yield the same bounds. Here they differ, and since we are interested in the upper bound we simply pick whatever expression is the greatest depending on the constants:

\[
D_{i,j} \leq D_{i,j-1} + \max \left\{ \text{gap}, -\text{gap} + \text{cmp}_{\text{max}} \right\}
\]  

(7.19)

This confirms the encoding hypothesis given in 7.12.

7.5 Finding internal extreme values

Finding the minimum and maximum values inside the \( t \)-block is important for the Four-Russians algorithm when using local alignment, for two reasons:

1. The lowest value is directly related to the number of subtables that must be used for a given scoring system and \( t \)-block size, as described in under Problem 3 in chapter 6.
2. When encoding the maximum relative value of a \( t \)-block, we need to know that enough bits are allocated to hold this number in each table entry. See Problem 2 in chapter 6.

This section will concentrate only on local alignment since there is no point when doing global alignment. Finding the minimum and maximum values depends on the following parameters: \( \text{cmp}_{\text{max}}, \text{cmp}_{\text{min}}, \text{gap}, t_n \) and \( t_m \). \( I_{\text{max}} \) and \( I_{\text{min}} \) are also needed, but can be computed using the equation given in 7.12.

One way to do it, for the minimum value, is to set all input cell jumps to \( I_{\text{min}} \) and assume every comparison to return \( \text{cmp}_{\text{min}} \). Then simply compute the \( t \)-block and record the lowest value relative to \( A \). The zero-case in Smith-Waterman should be ignored for now, since the lowest relative value is to be found. This is also the main idea in the following subsections. If all inputs and comparison yield the lowest possible value, we must also get the lowest internal value possible.

Since all inputs are using the same offset value between cells the extreme values are the same for \( t_{a \times b} \) and \( t_{b \times a} \) where \( a \) and \( b \) are sizes of either the width or height of the block. In case the block is symmetric computing only the diagonal and the cells to one side will give the same result as computing all the cells. We still have to be careful to compute the remaining part if \( t_n \neq t_m \). See figure 7.2.

7.5.1 Internal minimum - \( \text{V}_{\text{min}} \)

One might think that we need to find the lowest internal value relative to the lowest input edge value, since we can only get valid inputs. For instance, the algorithm will never see \( A = 2 \) and an input edge that goes to \( -3 \). This clearly goes against the local recursion’s zero-case. Finding the minimum relative value to the lowest input edge and to \( A \) are two different things, where the relative value to \( A \) is a lower-bound for the relative value to the lowest edge. This section will try to find the minimum value inside any \( t \)-block relative to \( A \).
7.5. FINDING INTERNAL EXTREME VALUES

Figure 7.2: When the input to either axis is the same, the cell values in the gray region will be identical to those mirrored in the diagonal. The gray region is irrelevant as far as finding internal extreme values is concerned. For index $T_{i,j}$ where $i < j$, index $T_{j,i}$ is a duplicate value and can be ignored.

It is of course better to have a constant time expression which can be computed at compile time instead of naively computing half a $t$-block in order to find $V_{\text{min}}$. Counting the edges from $A$ to the lower-right entry, $T_{i,j}$, and assuming $I_{\text{min}}$ for each jump gives a lower-bound for the minimum value. $V_{\text{min}} \geq I_{\text{min}} \cdot (t_n + t_m - 1)$. But just because a cell can jump $I_{\text{min}}$ does not necessarily mean it will. Especially not in a confined context of a $t$-block where the cells depend on each other. Remember that $I_{\text{min}} = \text{gap}$ from (7.12), so every input, in the minimum case, jump $\text{gap}$ distance. Let $V_{\text{min}}$ be the lowest value in the $t$-block, including the edges, relative to $A$. Let $t_{\text{min}} = \min(t_n, t_m)$ and $t_{\text{max}} = \max(t_n, t_m)$.

Case $\text{gap} \geq 0$: All input cells will take a value $\geq 0$ because $\text{gap} \geq 0$ and $\text{gap}$ is used for the input jumps. Since local alignment uses the max-function, all internal cells will also be $\geq 0$ because they are a result of an input cell plus $\text{gap}$. The minimum value of a $t$-block in this scoring system is $A$ itself. $V_{\text{min}} = 0$.

Case $\text{cmp}_{\text{min}} \geq \text{gap}$: $T_{1,1} \geq T_{0,1}$ because $\text{cmp}_{\text{min}} \geq \max(2 \cdot \text{gap}, \text{cmp}_{\text{min}})$. $T_{1,1}$ will thus have a value bigger than the cell above, $T_{0,1}$. This also hold for the rest of the cells in row 1. The jump between $T_{1,j}$ and $T_{1,j+1}$ for $j = 1..m - 1$ is $\text{gap}$, which is the same as the input edge. The property that $T_{i,j} \geq T_{i-1,j}$ hold for any row, considering the relevant cells in figure 7.2. This means the lowest value is in the longest input edge. Minimum is $V_{\text{min}} = t_{\text{max}} \cdot \text{gap}$.
Case \( \text{cmp}_{\min} \leq 2 \cdot \text{gap} \): In this case the gap-case will be taken for any cell due to the case inequality and because the input cells grow with \( \text{gap} \). Since \( \text{gap} < 0 \), due to the first case, the minimum value must be at the lower-right corner of the \( t \)-block. Minimum is \( V_{\min} = (t_{\min} + t_{\max} - 1) \cdot \text{gap} \). The same score is obtained by diagonally zig-zagging through the block and jumping the remaining gaps in the asymmetric sizes. Let the (potentially) asymmetric padding be \( \delta = t_{\max} - t_{\min} \). The minimum is also given by \( V_{\min} = 2 \cdot \text{gap} \cdot t_{\min} + \delta \cdot \text{gap} \) for this case.

Case else (\( \text{cmp}_{\min} < \text{gap} \)): The \( (T_{0,0} + \text{cmp}_{\min}) \)-branch is taken for \( T_{1,1} \), because the \( \text{cmp}_{\min} > 2 \cdot \text{gap} \), due to the previous case. The value to the right of \( T_{1,1} \) is increased by \( \text{gap} \) either by using \( T_{1,1} = \text{cmp}_{\min} \) or by adding \( \text{cmp}_{\min} \) to the second cell in the input row, which has value \( \text{gap} \). The point is that \( T_{i,j+1} = T_{i,j} + \text{gap} \). We can take the \( \text{cmp}_{\min} \) path a total of \( t_{\min} \) times. For asymmetric \( t \)-sizes the minimum become smaller by following the final \( \delta \) gaps since \( \text{gap} < 0 \). The minimum for this case is \( V_{\min} = \text{cmp}_{\min} \cdot t_{\min} + \delta \cdot \text{gap} \).

If \( T_{i,j} \) has neighbors up and left with value \( \text{gap} \) and diagonal neighbor with some value \( x \), which is often in the above cases, the value of \( T_{i,j} \) is given by \( \max(\text{cmp}_{\min}, 2 \cdot \text{gap}) \). This means that the two last cases can be combined. A more compact formula for the internal minimum value relative to \( A \) is given below:

\[
V_{\min} = \begin{cases} 
0 & \text{if } gap \geq 0 \\
\text{gap} & \text{if } \text{cmp}_{\min} \geq \text{gap} \\
\max(\text{cmp}_{\min}, 2 \cdot \text{gap}) \cdot t_{\min} + (t_{\max} - t_{\min}) \cdot \text{gap} & \text{else}
\end{cases} \quad (7.20)
\]

The above function has been verified by running a large number of combinations of \( \text{cmp}_{\min} \), \( \text{cmp}_{\max} \), \( \text{gap} \), \( t_{\max} \) and \( t_{\min} \) through a script, and comparing to the value obtained by computing the \( t \)-block naively.

### 7.5.2 Internal maximum - \( V_{\max} \)

The number of bits used in each index is partially decided by \( V_{\max} \). The exact value is nowhere near as important as \( V_{\min} \). An upper bound is sufficient. Assuming that each jump is \( I_{\max} \), the highest value resides in the lower-right corner of the \( t \)-block, and is given by \( V_{\max} \leq \max((t_n + t_m) \cdot I_{\max}, 0) \). The flooring max-function is included in case \( I_{\max} < 0 \).
Chapter 8

Speeding up LUT construction

It seems that most articles dealing with Four-Russians quietly step over the matter of generating
the lookup table for the block function. Constructing a LUT for a large block size can be rather
time consuming when using a naive algorithm for some scoring systems. This chapter presents a
fairly simple algorithm for quickly constructing a LUT for both global and local alignment with
arbitrary scoring systems. Global alignment is considered first, and the algorithm is updated to
work with local alignment later in the chapter.

The simplest way of computing the lookup table is to iterate all the input combinations and
calculate the encoded output by solving the block as a normal \( t \times m \) alignment problem naively.
This is the approach was chosen for the naive construction scheme in this thesis. One could also
iterate all the LUT entries and decode the index into the corresponding input information for the
\( t \)-block. The decoding does add some overhead. Especially when the encoding is not completely
optimal, i.e. when 2 bits are used to store an interval of 3, like unit cost.

The presented method is rather straightforward when looking at the two other techniques
it is based on: Depth-first traversal and divide-and-conquer. A \( 3 \times 3 \)-block is used as basis for
examples in the this chapter, unless otherwise stated, but the technique generalizes to other
asymmetric sizes as well.

8.1 Depth-first search (DFS)

Many partial \( t \)-block calculations invariably end up being repeated in the in the naive approach,
since some cells in the body of a given \( t \)-block will share similar values with other blocks. This
happens because all combinations of the encoding are computed, and a small change in the
encoding will often only yield a slightly different result. An example being the two \( t \)-blocks with
identical encodings except for \( C_2 \). See figure 8.1. It is easy to see that the first two rows will be
the same, since \( C_2 \) can only affect the cells in the third row.

Iterating all the combinations can instead be seen as a depth-first search, where each cell
in the \( t \)-block corresponds to a level in the search. Identical partial solutions are reused since
children of that node builds on this partial solution. The only problem is that some cells are
restricted by the choices of input in earlier steps.

Consider the calculation in figure 8.2. All the required parts of the encoding has already been
decided before calculating the cells prior to the one marked in red. Therefore the cell can only
have one value, 1. In this case it is not a big problem, it just means that this particular node,
among others, will only have one child in the traversal. Using this strategy our recursion tree
will have a depth of \( n \times m = 9 \), one for each cell in the body of a \( 3 \times 3 \)-block.
We observe that only the cells which depend on a new part of the encoding will spawn several combinations in its list of branches. $X_iC_i$ and $Y_jB_j$ go hand in hand since it is not possible to compute new cells using only one part of either, when advancing a row or column.

The solution is to either add combinations for $X_iC_i$ or $Y_jB_j$ for each node, where $i$ and $j$ is the index of the next row or column, respectively, and calculate as many cells as possible for that node before moving on. This also reduces the recursion depth to $1 + (t_n - 1) + (t_m - 1) = t_n + t_m - 1 = 5$.

We have to start at $T_{1,1}$ since we obviously cannot compute a cell whose value depends on a cell whose value has not been calculated yet. This cell’s value depends on encoding $Y_0$, $B_0$, $X_0$, and $C_0$. This gives $(I_1|\Sigma|^2 = 144$ branches for this first cell. Moving one level down in the recursion we add $Y_1B_1 = X_1C_1 = I_1|\Sigma| = 12$ new combinations. This can be done a total of $t_n - 1 + t_m - 1$ times. Thus, we end up with $144 \cdot 12 \cdot 12 \cdot 12 = 2,985,984$ combinations in total for our example, which matches the number from table 3.1.
8.1.1 Number of function invocations

Theoretically, all the work performed in the algorithm is evaluating the \(\min\)-function given by recursion (2.1). The naive approach will invoke this function a total of \(t_n t_m = 9\) times for each entry in the lookup table. (If we disregard the time spend on initializing the A, B and, C-blocks.)

\[
\frac{3}{12^0} + \frac{2}{12^1} + \frac{2}{12^2} + \frac{1}{12^3} + \frac{1}{144^4} = 3.1811 \quad (8.1)
\]

\[
\frac{3}{12^0} + \frac{3}{12^1} + \frac{1}{12^2} + \frac{1}{12^3} + \frac{1}{144^4} = 3.2575 \quad (8.2)
\]

Equation (8.1) is the amortized number of function invocations performed for each entry in the depth first traversal, using the left strategy in figure 8.3. Equation (8.2) is for the right strategy.

Consider the left approach in figure 8.3 and equation 8.1. The last level will always have to evaluate the recursive function three times. The level before will have to evaluate the function two times every 12th index, \((I|\Sigma)|\), and the level before two times every 12\(^4\)th time, and so on. The left strategy should, in theory, be a bit faster. The “hard three” seems impossible to avoid since the last row is always either \(t_n\) or \(t_m\) in width. Both strategies almost achieve a factor three improvement on the naive approach and even better on larger values of \(t_n\) and \(t_m\).

8.2 Divide and Conquer

The other idea is to generate a lookup table of smaller \(t\)-blocks in order to quickly populate the real table.

![Figure 8.4: A \(t_{4\times4}\)-block can be computed using four lookups from a single \(t_{2\times2}\)-table.](image)

Optimally, we only want to create a single intermediate lookup table for each level in the recursion and reuse that several times for each block. For odd \(t\)-sizes, one could use a flat \((t_{1\times t_m})\) intermediate table, and “sweep” the \(t\)-block instead of divide and conquer, by performing \(t_n\) lookups in the intermediate table. An algorithm for computing a LUT using divide and conquer is given in figure 8.5.

This of course works best if \(t_n\) and \(t_m\) are powers of two since only one intermediate table is needed for each recursive step. If a \(t_{1\times1}\) LUT is already computed \(((I|\Sigma))^2 = 144\) function evaluations) the number of lookups performed per entry by the recursive steps is, when \(t = 4\):
Create LUT_{i−1} for size t/2 // Can recursively use the pseudo-code below.

for every entry idx in LUT_i {
  y, b, x, c = encoding(idx)
  upperL = LUT_{i−1}(y_{0..t_m/2}, b_{0..t_m/2}, x_{0..t_n/2}, c_{0..t_n/2})
  upperR = LUT_{i−1}(y_{t_m/2..t_m}, b_{t_m/2..t_m}, x_{0..t_n/2}, upperL.C')
  lowerL = LUT_{i−1}(y_{0..t_m/2}, upperL.B', x_{t_n/2..t_m}, c_{0..t_n/2})
  lowerR = LUT_{i−1}(y_{t_m/2..t_m}, upperR.B', x_{0..t_n/2}, lowerL.C')
  LUT_i(idx).B' = concat(lowerL.B', lowerR.B')
  LUT_i(idx).C' = concat(upperR.C', lowerR.C')
}

Figure 8.5: Pseudo-code for generating the main lookup table recursively using temporary tables of smaller t-blocks.

\[
\sum_{i=1}^{\log_2(t)} 4(\lvert I\rvert\lvert \Sigma\rvert)^{2^i} (\lvert I\rvert\lvert \Sigma\rvert)^{2^t} = 4.0001929 \tag{8.3}
\]

Each level creates a new table which is \((\lvert I\rvert\lvert \Sigma\rvert)^{2^i}\) times bigger than the previous level. The number of lookups is dominated by the last level, which performs four. Since the last level is the main decider for the number of lookup, it might make more sense to split in two, with the cost of more recursive levels. We get

\[
\sum_{i=1}^{\log_2(t)} 2 \left( (\lvert I\rvert\lvert \Sigma\rvert)^{2^i} + (\lvert I\rvert\lvert \Sigma\rvert)^{2^{i-1}} \right) (\lvert I\rvert\lvert \Sigma\rvert)^{2^t} = 2.013986 \tag{8.4}
\]

lookups per entry. However, twice as many recursive LUTs will have to be created, and it still only works elegantly when both \(t_n\) and \(t_m\) are a powers of two.

8.3 Combining the ideas

Combining the idea means we use the DFS-approach, but use small intermediate lookup tables in each node to compute the children, instead of naively calling the recursive function. Looking at the right part of figure [S.3], it is easy to see that this path would be best here, since only two different intermediate tables are needed. One \(t_{1\times1}\) for the first row and one \(t_{1\times3}\) for the downwards sweep. The left approach would need four different tables, or three if extra work is performed to convert the encoding from \(D_3 \rightarrow D_4\) to lookup in the table used in step \(D_2 \rightarrow D_3\).

In order to reuse and keep the number of intermediate lookups low, the best strategy seem to be to split computation into two steps:

1. Create \(t_{1\times t_m}\) "flat" intermediate table. For practical sizes it is sufficient to simply use the DFS strategy from the right part of figure [S.3] since the Four-Russians experiments show that \(t_{1\times1}\) has no significant improvement over naive. Also, the creation on this table is rather fast compared to the full \(t_{n\times t_m}\) computation. It is \((\lvert I\rvert\lvert \Sigma\rvert)^{t_n}\) times faster.
2. Using the \( t_1 \times t_m \) -table we can either "DFS-sweep" our way to the full \( t_n \times t_m \) table with depth \( t_n \) or use it as a base for creating the \( t_2 \times t_m, t_4 \times t_m, \ldots, t_{n/2} \times t_m \) intermediate tables. Of course only \( t_{n/2} \times t_m \) would be used for the final computation, the ones prior are only used for quickly creating \( t_{n/2} \times t_m \). For practical purposes a sweep will likely be best, since dividing only makes sense for \( t_n \geq 4 \).

The table should ideally be filled as sequentially as possible in order to run fast on modern processors. This is partially possible. (Even fully, see section \[8.3.2\].) In order to do that the most significant part of the index should be iterated in the outer loops. This means iterating \( Y \) of \( YBXC \) first. Like when fixing \( Y \) for a column-computation in the Four-Russians algorithm, fixing \( Y \) in the outer loop when computing the table also have a nice property: Instead of computing the flat table, \( F_{\text{flat}}(Y, B, X_1, C_1) = B', C'_1 \), we can compute \( F_{\text{flat}}(B, X_1, C_1) = B', C'_1 \) for every value of \( Y \). \( Y \) stays fixed and does not need to change when sweeping downwards since all consecutive computations will depend on that \( Y \) unlike \( B \) which changes for each level. Like the naive construction, this also makes multithreading the code trivial, since each value of \( Y \) is computed independently.

The algorithm for the sweeping-version of the algorithm is given in simplified pseudo-code in figure \[8.6\].

```plaintext
// Iterate all combinations of YBXC and generate B' and C' for each:
foreach value y of Y {
    Compute \( F_{\text{flat}}(B_{1..t_m}, X_1, C_1) \) according to y
    foreach value b of B {
        function DFS(level, encoding, B', C') {
            if level == t_n {
                LUT[encoding] = B', C' // Base case. Write entry.
            } else {
                foreach x, c in \( \Sigma \times I \) {
                    // Combinations for new row.
                    row = F_{\text{flat}}(B', x, c)
                    DFS(level + 1,
                        concat(encoding, x, c),
                        concat(C', row.C')) // Accumulate output column.
                }
            }
        } // function DFS
        DFS(0, concat(y, b), B)
    }
}
```

Figure 8.6: Pseudo-code for quickly generating lookup tables used by the Four-Russians method.

Moving the computation of \( F_{\text{flat}} \) to inside the outer loop reduces its space consumption by a factor \( |\Sigma|^m \). This is not much in the grand scheme of things, but it may make the algorithm run faster. A big part of the table construction is computing the encoding. The DFS-approach not only accumulates the result but also the encoding. The experiments will show how much significance this has.
8.3.1 Intermediate table - $F_{\text{flat}}$

The number of lookups performed in $F_{\text{flat}}$ for each final entry computed in the final LUT is

$$\sum_{i=0}^{t_n} \frac{1}{(I|\Sigma|)^i}$$

(8.5)

For instance, when $|\Sigma| = 4$, using unit costs, which has $|I| = 3$, and $t_n = 3$ the amortized number of intermediate lookups per entry becomes

$$\frac{1}{12^0} + \frac{1}{12^1} + \frac{1}{12^2} = 1.0903$$

(8.6)

The effect of the later terms, as $t_n$ grows, quickly diminishes. The number of lookups in $F_{\text{flat}}$ can be bounded as a geometric series, and for a given choice of $I$ and $\Sigma$ the time taken to compute a single final entry is essentially constant:

$$\lim_{t_n \to \infty} \sum_{i=0}^{t_n} \frac{1}{(I|\Sigma|)^i} = \frac{I|\Sigma|}{I|\Sigma| - 1} = O(1)$$

(8.7)

Disregarding the construction time of $F_{\text{flat}}$ this is quite the theoretical improvement over the naive algorithm which will use $O(t_nt_m)$ time per entry.

8.3.2 Sequential memory access

As we have seen in the results from figure 5.3, cache locality is very important. The assumption is that sequential reads and writes are also important when dealing with data that does not fit into the CPU’s cache.

The saving grace in the naive approach is that each entry in the table is calculated independently from each other. This means that the memory can be filled in a sequential manner. However, this is not quite as simple when the subcalculations start to depend on each other.

The method presented in section 8.3 does partially work in a sequential fashion in that both the Y and B component of the index, and the first part of X are iterated in order. These parts make up the most significant bits of the index, so the indices can be iterated roughly in order. The problem presents itself when we try to iterate X and C in order. $X_i$ and $C_i$ are paired since picking a new encoding for either does not give enough information to compute a new row. If we want the 1.0903 intermediate lookups per index we have to iterate in pairs of $X_iC_i$, as seen in figure 8.7.

![Figure 8.7: The iteration pattern of the algorithm given in figure 8.6.](image)

This will obviously not produce a sequential write pattern since $C_0$ is fixed in the outer loop and $X_1$ and $X_2$ will be iterated in the inner loop. The only solution, if we truly want sequential writes, is to forgo the pairing of $X$ and $C$ and iterate $X$ in the outer loop, while only using the combined trick on $C$ itself. This does mean slightly more intermediate table lookups. We only
have to iterate \( I \) combinations per level instead of \( I|\Sigma| \), but this work has to be repeated for every combination of \( X \)'s encoding. The number of intermediate table lookups per index is thus

\[
\sum_{i=0}^{t_n} \frac{1}{p_i}
\]  

which, for the example, is \( 1 + \frac{1}{3} + \frac{1}{3} = 1.4444 \). The cost for sequential memory writing is approximately \( 1/3 \) more intermediate table lookups per final entry.

### 8.4 Local alignment

It is not hard to update the algorithm to work with local alignment as described in chapter 6. First off, every entry needs to have an extra \( \text{max} \)-field, containing the highest value in the block relative to \( A \). This is simply a matter of propagating the value of \( A \) through each row, as seen in figure 8.8. The current maximum is given by

\[
\begin{align*}
  \text{max}_0 &= -\infty \\
  \text{max}_i &= \max(\text{max}_{i-1}, A_i + F_{\text{flat}}(B', x, c).\text{max}) \quad \text{where} \quad i \in [1..t_n]
\end{align*}
\]

\( \text{max}_i \) is the maximum value between rows \( 1..i \). Of course, \( F_{\text{flat}} \) must also be updated to contain a \( \text{max} \)-field for each entry, and \( V_{\text{min}} \) must be found for the flat table in order to know how many distinct values of \( A \) must be handled when calling \( F_{\text{flat}} \).

![Figure 8.8: Updating A and the max value for each level in the DFS.](image)

In addition to computing the \( \text{max} \) field, it should be possible to create a LUT for a given value of \( A \). This detail goes hand in hand with the above requirement, since \( A \) has to be propagated through the computation anyway. The LUT build procedure should simply take an extra argument containing the value of the initial \( A \).

### 8.5 Verifying correctness

The naive building procedure is assumed to produce a correct lookup table. The fast algorithm is checked for correctness simply by building the lookup table and comparing every entry to the corresponding entry in a table built by the naive algorithm.

For local alignment, some tables built with a small value of \( A \) may contain invalid entries. For instance, when \( B \) or \( C \) is \([-1, -1, -1]\) and \( A = 2 \), resulting in a cell with value \(-1\). This scenario is never possible due to the zero-case. The local alignment variant of Four-Russians will also never access these entries. However, since the tables contain all combinations of \( YBXC \) and \( A \)'s, some invalid entries exist. This is perfectly OK. They should just be ignored when comparing two tables from different algorithms, since they may generate these invalid entries differently.
8.6 Experiments

Only the speed taken to generate the tables is relevant. The methods should all produce identical tables, if implemented correctly. It has already been determined that the naive construction algorithm produces a correct table, because the Four-Russian method has been run against the naive algorithm on a very large random input and yields the same result. Tables produced by the methods were verified to be identical. Four construction variants are tested:

- **Normal**: This is the naive version which will calculate each entry as a separate $t$-block.
- **Fast**: The improved version from section 8.3. $F_{flat}$ was sufficiently small to be put on the stack.
- **Fast, Sequential**: The attempt at a sequential memory fill, at the cost of more intermediate lookups from section 8.3.2.
- **Fast, Full Y**: Rather than computing $F_{flat}$ inside the outer loop with $Y$ as an implicit parameter, $F_{flat}$ is computed for any $Y$ in a completely separate step, using additional memory.

**Memory overhead**: This method simply fills the table with the value 123. It is therefore not a correct solution. It is merely used as a reference for how fast it is possible to fill the table with no computing cost, only considering the write speed of the memory. Note that unit cost edit distance does not completely fill all the allocated entries in the table because $|I| = 3$ and two bits are used. The dummy implementation simply fills all entries in order to avoid encoding overhead. Simply filling an array with size of the actual usage is not fair either, since the real table have holes scattered throughout, which likely worsens performance compared to a completely sequential fill.

Each construction algorithm was run five times and the best time was recorded. Neither the array allocation or subsequent free-call were part of the recorded run-time. Before running the construction algorithms, all the table entries were initialized to zero in order to force the operating system to commit the allocated pages to physical memory.

8.7 Results

As seen in figure 8.9 all versions of the improved algorithm are faster than the naive approach. Computing $F_{flat}$ with $Y$ as an explicit parameter was never expected to be faster, it was included to see the performance-penalty of using that extra memory in the flat table. As seen in the figure, it is slower than the normal fast algorithm for any $t$.

The extra cost of the sequential variant is not worth it in most cases. It only seems to make sense when $t_n$ is smaller than $t_m$. This is likely because the depth of the DFS is shallow in these cases and the extra cost is not felt as much.

As expected the the gap between the naive and fast algorithm becomes even bigger when computing the bigger $t$-sizes. This is expected because the naive algorithm does $O(t_n t_m)$ work per entry whereas the fast algorithm’s workload per entry is roughly independent of $t$. The best case improvement happens in local alignment, $t_{4 \times 3}$ when the fast algorithm is around 18 times faster than the naive. It is likely even faster for bigger sizes, but $n + m = 7$ requires 1GB of RAM for just one table. Incrementing either $n$ or $m$ would require 16GB of RAM.
8.7. RESULTS

![Graph showing speedup ratio vs naive for global and local alignment using different variations of the new fast algorithm compared to the naive.](image)

Figure 8.9: LUT construction performance for global and local alignment using different variations of the new fast algorithm compared to the naive.
CHAPTER 8. SPEEDING UP LUT CONSTRUCTION

8.7.1 Overhead

Although $t_{3 \times 4}$ and $t_{4 \times 3}$ create equally sized tables, the fast algorithm will take longer to compute $t_{3 \times 4}$. Consider $t_{n \times 4}$ where $n \in [1..3]$ in figure 8.10. $m = 4$ is the width of the block and the computation of the flat table is the same for all cases. In the case of $t_{1 \times 4}$ a flat table of width 4 is created and the main computation essentially writes the result to the main table since the base case in DFS is reached instantly, which is also why there is almost no "Main computation" cost. Notice that $t_{3 \times 4}$ and $t_{4 \times 3}$ almost run equally fast except for the computation of $F_{\text{flat}}$. $F_{\text{flat}}$ has width 4 in $t_{3 \times 4}$ and 3 in $t_{4 \times 3}$. However, $t_{3 \times 4}$ only has to recurse three levels in the DFS, which amounts for

$$\sum_{i=0}^{t_n} \frac{1}{(I|\Sigma|)^i} = \frac{1}{16^0} + \frac{1}{16^1} + \frac{1}{16^2} = 1.0664$$ (8.10)

amortized lookups per entry, while $t_{4 \times 3}$ has four levels and thus

$$\frac{1}{16^0} + \frac{1}{16^1} + \frac{1}{16^2} + \frac{1}{16^3} = 1.06665$$ (8.11)

lookups per entry. Hardly any difference in the main computation cost, which is reflected in the results. It should also come as no surprise after looking at equation 8.7. The main computation part of $t_{4 \times 3}$ is actually a bit faster than $t_{3 \times 4}$. (When disregarding the flat computation.) This might be due to the smaller memory footprint of $F_{\text{flat}}$.

The irony is that Four-Russians works best in a column major fashion, with the current layout, which favors flat $t$-blocks, while the computation of the tables is fastest for narrow $t$-sizes. We could of course transpose the orientation of computing the tables, such that $F_{\text{flat}}$ is a column, but this would most certainly kill the performance since the memory access pattern would be wildly scattered.

It is worth noting that the memory overhead is about 20% of the total computation time in the fastest case. This means the algorithm is only five times slower than the optimized memset function.

![Figure 8.10: Time distribution for different tasks when filling a local alignment $t_{n \times 4}$-table. The memory overhead is obtained by running the dummy-method separately.](image)

The fast algorithm only makes slightly more than one amortized lookup in $F_{\text{flat}}$ for each entry in the main table. The encoding is also incrementally created in the DFS. The naive
approach runs in $O(t_nt_m)$ time per entry, but that does not account for the overhead of creating the encoding. In figure 8.11 we see that computing the encoding is a substantial part of the total computation time. The encoding overhead itself is several times bigger than the entire fast algorithm. But even when disregarding the encoding cost, the actual evaluation of the recursive function is still the biggest performance hit. No real difference in the computation times for $t_{3\times4}$ and $t_{4\times3}$ exist, neither in theory or practice.

The generated assembly for the naive implementation seems to be well-optimized. The CMOV (conditional move) instruction is emitted throughout the recursive function instead for a series of branches. This means there are no branch misses in the recursive function itself which would otherwise be expected, since the best neighbor almost never stays the same. A quick performance analysis confirms this hypothesis, with a branch-miss ratio of 0.31% for five runs of the naive algorithm.

![Figure 8.11: Time distribution for the naive algorithm for local alignment LUT creation. Encoding overhead was measured by running a modified naive algorithm with no actual computation.](image-url)
Chapter 9

Local alignment experiments

This chapter is very similar to the chapter about global alignment experiments in structure. The theory from the previous three chapters is tested in practice here. Some of the same experiment from global alignment are run again for local alignment, if they are relevant. Some experiments specific to local alignment are obviously also tested here. As in previous experiments, all plots with input length as the x-axis are plotted on a logarithmic scale.

Two scoring systems are tested for local alignment. A default system which does not require very much space compared to edit distance and an extended system used for testing Four-Russians in a more realistic setting.

Default scoring system

\[ \text{score}(\alpha, \beta) = \begin{cases} 1 & \text{if } \alpha = \beta \\ -1 & \text{else} \end{cases} \]
\[ \text{gap} = -1 \] (9.1)

Using encoding formula 7.12 we get the interval

\[ I = [-1...\max(-1, 1 - (-1))] = [-1...2] \] (9.2)

which means each encoding fit perfectly in 2 bits.

Extended scoring system

\[ \text{score}(\alpha, \beta) = s_{\alpha, \beta}, \text{ where } s = \begin{pmatrix} 1 & 0 & -1 & -2 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 3 & -1 \\ -2 & -1 & -1 & 1 \end{pmatrix} \]
\[ \text{gap} = -2 \] (9.3)

The choice for entries in \( s \) are totally made up and are unlikely to have any significance in a real-world scenario. The extreme values have been picked in order to not create any waste in the encoding unlike edit distance, and also not to be too positive such that the zero-case is hit more often. The interval is thus

\[ I = [-2...\max(-2, 3 - (-2))] = [-2...5] \] (9.4)

which gives \( |I| = 8 \) different jump values that fit into 3 bits without waste.
9.1 Four-Russians and \( t \)-block sizes

While the number of allocated entries in a lookup table for local alignment using the default scoring system is the same as for edit distance, the number of used entries have increased by a factor \((4/3)^{t_n+t_m}\). In addition to that, local alignment will also need several tables in order to handle the zero-case in the recurrence. See section 6.4. This experiment is set up like the test in section 5.4 in order to find out which \( t \)-size is best for a given input size.

Result

The results are in figure 9.1 almost look identical to the same setup using global edit distance. The size of a table of the default local alignment scoring system is the same size as the table for edit distance. The only difference is that there is no waste in case of local alignment. It was determined in section 5.6 that the size of entire table, even with waste, was likely more important than the number of used entries in it. By that logic, the local alignment should not be that much slower. The main difference here is that more than one table is needed to handle the zero-case. In table 9.1 an overview over the memory used for each table size, and how many tables are needed in order to handle the zero-case, is given.

Like in the global alignment test, this test was also run with a smaller fixed size of \( m \) and a very large \( n \) to find out if, and when, the bigger \( t \)-blocks converge towards a constant speedup. It turns out also to be the case here, although \( t_{3\times4} \) does not get the same massive speedup as it does in edit distance. In table 9.2 is given an overview of the cache misses. As opposed to edit distance the number of misses grow with \( n \), which is likely the reason the speedup is nowhere near as large. However, it does not explain why the number of misses grow with \( n \), but the performance still becomes better.

Despite the increase in memory usage, the conclusion is similar to the one for global alignment, albeit with smaller values of \( t \). \( t_{2\times2} \) and \( t_{2\times3} \) seem like the best choices with \( t_{2\times4} \) becoming relevant for large input.

Because \( Y \) of \( YBXC \) is locked for most of the computation, the memory access area becomes quite small, even for \( t_{2\times4} \). \( 320MB/|\Sigma|^{t_m} = 1.25MB \), which is the memory accessed in the LUT for a given column. However, since there are five tables the access pattern is split into five non-contiguous regions. It might have given a small boost to use a specialized local index layout of \( YABXC \) or something else where \( A \) is less significant than \( Y \).

<table>
<thead>
<tr>
<th>( t )</th>
<th>#tables</th>
<th>Table size (KB)</th>
<th>Total memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_{1\times1} )</td>
<td>1 + 1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>( t_{1\times2} )</td>
<td>1 + 2</td>
<td>16</td>
<td>48</td>
</tr>
<tr>
<td>( t_{2\times2} )</td>
<td>1 + 2</td>
<td>256</td>
<td>768</td>
</tr>
<tr>
<td>( t_{2\times3} )</td>
<td>1 + 3</td>
<td>4,096</td>
<td>16,384</td>
</tr>
<tr>
<td>( t_{3\times3} )</td>
<td>1 + 3</td>
<td>65,536</td>
<td>262,144</td>
</tr>
<tr>
<td>( t_{2\times4} )</td>
<td>1 + 4</td>
<td>65,536</td>
<td>327,680</td>
</tr>
<tr>
<td>( t_{3\times4} )</td>
<td>1 + 4</td>
<td>1,048,576</td>
<td>5,242,880</td>
</tr>
</tbody>
</table>

Table 9.1: Memory usage for lookup tables in local alignment.
Figure 9.1: Local alignment with Four-Russian using different sizes of $t$ and a separate LUT for each of the special $A$-cases. $V_{mn} + 1$ tables are needed in total.

Figure 9.2: Same setup as figure 9.1 except $m = 10,000$ and $n$ goes up to 21 millions. The $t$-blocks that reach a constant speedup at small input size were omitted.
9.2 $t$-block sizes with naive fallback

Since quite a bit of extra memory is used in order to handle the zero-case, a hypothesis is that it may make more sense to compute such cases naively. Depending on how much these subtables are used, it might not be worth the overhead of using additional memory. The problem with mixing Four-Russians with the naive version is the conversion back and forth between the two representations. For a naive fallback the input edges will have to be written back into a matrix of values, and the output of the matrix must be encoded back into $B'$, $C'$ and max. This experiment tests the performance of local alignment Four-Russians when the cases where $A < -V_{\min}$ are handled naively.

The default uniform sampling was using to generate input sequences, resulting in 25% similarity. The amount of saved space can be derived from the table 9.1.

![Figure 9.3: Ratio of using a single lookup table with naive fallback versus a complete table for each of the subcases ($A < -V_{\min}$). Values above 1 means naive fallback is faster.](image)

<table>
<thead>
<tr>
<th>$n$</th>
<th>L1 misses</th>
<th>Approx L2 misses</th>
<th>L3 misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>5.64%</td>
<td>21.54%</td>
<td>54.12%</td>
</tr>
<tr>
<td>1,000,000</td>
<td>7.25%</td>
<td>62.41%</td>
<td>33.46%</td>
</tr>
<tr>
<td>10,000,000</td>
<td>8.35%</td>
<td>78.48%</td>
<td>27.91%</td>
</tr>
</tbody>
</table>

Table 9.2: Cache misses for $t_{3\times4}$ at differently sized inputs. $m = 10,000$. 
CHAPTER 9. LOCAL ALIGNMENT EXPERIMENTS

Result

See figure 9.3. Small blocks, $t_{1 \times 1}$, $t_{1 \times 2}$ and $t_{2 \times 2}$ seem to favor the full table strategy no matter the input size because the tables are so small that a several factor increase in memory usage is not going to change anything.

For the rest of the $t$-sizes the small advantage of naive fallback quickly begins to fall off when the input size grows. Handling the zero-cases naively does not seem like a good idea here. Only $t_{2 \times 3}$ seems to benefit briefly. The hypothesis was that the blocks with $A < -V_{\text{min}}$ were rare compared to the general blocks. This does not look to be the case, at least not with the default input similarity.

9.3 Subtable usage

The previous experiment showed that the blocks where $A < -V_{\text{min}}$ also need to be computed quickly. Input similarity should have some influence on how much different lookup tables are used. We can test this hypothesis by counting the number of lookups performed on each table, under a varying similarity percentage. The algorithm for generating a pair of input sequences is given in section 4.1.

![Figure 9.4: Usage of the different subtables in local alignment using a $t_{2 \times 3}$-block size. We know from equation 6.4 that three fallback tables are needed for this particular configuration. Nothing interesting happens at similarity percentages higher than 30 since they are dominated by LUT$_{A \geq 3}$. A pair of 40,000 long sequences were aligned in this plot.](image-url)
9.4. INPUT SIMILARITY EFFECT ON FOUR-RUSSIANS

Result

Uniform sampling of symbols in the input sequences converge towards a similarity of 25%. Looking at figure 9.4, we can see that around 25% the usage of the different tables are fairly evenly distributed. This would mean that generating a separate LUT for each case is likely the best idea. This also means that the idea for naive fallback is not very good when the match similarity is $\leq 25\%$, since $A$ dips below the $-V_{\text{min}}$ point quite often there. Testing naive fallback on other match similarities might make more sense.

Figure 9.5: Naively computing local alignment when $n = m = 2,000$. The actual input sequence pairs were sampled from the above base strings. Even though all pairs have a match similarity of 25% the score still differs a lot. The blue area represents cells whose values is $\leq 3$, and must be handled by the specialized tables in Four-Russians. The warmer colors represent higher cell values.

Looking at figure 9.5 we can see that even at the fixed similarity of 25% there is still some fluctuation in the final score on different inputs, and of course the number of queries in the tables for $A < -V_{\text{min}}$. This would also explain the roughness in figure 9.4 around 22 – 26%.

9.4 Input similarity effect on Four-Russians

We have seen the effects of different $t$-sizes for varying input size with a fixed similarity percentage of 25%. The two most likely $t$-sizes are $t_{2 \times 2}$ and $t_{2 \times 3}$. They seem roughly equally fast around $n = 15,000$ when scoring input sequences sampled uniformly from $\Sigma$. This test is looking at the performance using input with varying match similarity. Also, the naive fallback idea is considered again since it might fare better with a certain type of input.

Result

The results are shown in figure 9.6. The plot does not show the fact that the performance of the naive algorithm is completely unaffected by the similarity of the input, as expected.

Naive fallback is slower than full tables when all hits goes to fallback due to the low similarity, while the actual computation remains the same. This was also to be expected.

The naive fallback should have an advantage against the full table version at max similarity, since only one table will be used. It turns out naive fallback is faster for $t_{2 \times 2}$ and slower for $t_{2 \times 3}$. Overall there does not seem to be any type of input where it is better to fall back to naive computation.
9.5 Preprocessing vs Four-Russians

The point of generating the lookup tables quickly is to improve the total running time of the program. This test is devised to find out, for both the naive and fast LUT construction scheme from chapter 8, when it is feasible to use the Four-Russians implementation over the naive, and how much better it is, when the total running time including preprocessing is considered.

Looking at the previous results it seems the Four-Russians’ speedup over the naive grows with $n$. And because the time taken to construct the LUT is constant independently of $n$, we can use binary search in order to find the exact point, $n$. There is a bit of variation in running times, which means the values of $n$ found in the tables below can never be completely accurate. Running the test several times show that the margin of error is below 1%.

![Figure 9.6: Speedup of local alignment Four-Russians using $t_{2 \times 2}$ and $t_{2 \times 3}$, with and without naive fallback on $A < -V_{min}$. $n = 15,000$.](image)

<table>
<thead>
<tr>
<th>Time(LUT(t) + Four-Russians(t, n)) ≤ Time(Naive(n))</th>
<th>$t_{1 \times 1}$</th>
<th>$t_{1 \times 2}$</th>
<th>$t_{2 \times 2}$</th>
<th>$t_{2 \times 3}$</th>
<th>$t_{2 \times 4}$</th>
<th>$t_{3 \times 3}$</th>
<th>$t_{3 \times 4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive LUT</td>
<td>113</td>
<td>307</td>
<td>1,243</td>
<td>7,117</td>
<td>39,313</td>
<td>66,179</td>
<td>223,665</td>
</tr>
<tr>
<td>Fast LUT</td>
<td>95</td>
<td>289</td>
<td>515</td>
<td>2,807</td>
<td>17,555</td>
<td>45,775</td>
<td>95,619</td>
</tr>
</tbody>
</table>

Table 9.3: At which input size $n$ the sum of the LUT construction and Four-Russians computation are getting faster than the naive algorithm.
9.5. PREPROCESSING VS FOUR-RUSSIANS

Figure 9.7: Speedups of various $t$-block sizes for different input sizes, $n$ and $m$, considering no preprocessing ( ), naive preprocessing ( ) and fast preprocessing from chapter 8 ( ).

![Graphs showing speedups for different block sizes](image)

Table 9.4: $\text{Time(LUT(t))} \leq \text{Time(Four-Russians(t, n))}$

<table>
<thead>
<tr>
<th></th>
<th>$t_{1\times1}$</th>
<th>$t_{1\times2}$</th>
<th>$t_{2\times2}$</th>
<th>$t_{2\times3}$</th>
<th>$t_{2\times4}$</th>
<th>$t_{3\times3}$</th>
<th>$t_{3\times4}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive LUT</td>
<td>35</td>
<td>350</td>
<td>2,125</td>
<td>11,883</td>
<td>44,929</td>
<td>23,275</td>
<td>141,433</td>
</tr>
<tr>
<td>Fast LUT</td>
<td>47</td>
<td>321</td>
<td>905</td>
<td>3,445</td>
<td>9,617</td>
<td>4,985</td>
<td>27,017</td>
</tr>
</tbody>
</table>

At which input size $n$ the LUT construction time is equal to the time it takes to compute the score using the Four-Russians implementation. Remember that the problem size is $O(n^2/t^2)$ when comparing the value of $n$ found for both algorithms.
CHAPTER 9. LOCAL ALIGNMENT EXPERIMENTS

Result

Figure 9.7 shows that, depending on the block- and input-size, the cost of preprocessing does not matter very much in the grand scheme of things. Considering $t_{2 \times 3}$, which has all around good performance; around $n \geq 25,000$ there is barely a difference in total execution time regardless of preprocessing scheme. Even less so for $t_{2 \times 2}$. The point where preprocessing becomes irrelevant, is at bigger input sizes for $t_{3 \times 4}$ and $t_{3 \times 4}$, as expected.

Table 9.3 shows when the total time for both the construction and computation is better than the naive, which obviously has no expensive preprocessing step. For small sequences, $n < 10,000$, the fast construction might make sense if the program is only run once. For the biggest table, $t_{3 \times 4}$, the fast construction makes sense, since it is a lot faster here than the naive, and the construction time is relevant, even for large sequences. Unfortunately, $t_{3 \times 4}$ is slower in local alignment for all the sizes tested, so there is no point in picking it over $t_{2 \times 2}$, $t_{2 \times 3}$ or even $t_{2 \times 4}$. Figure 9.2 show that when $t_{3 \times 4}$ converges towards a constant speedup, it is still slower than the other $t$-sizes. Even if it, hypothetically, were to become better at even more extreme input sizes the LUT construction overhead is certainly negligible, given the time the alignment would take to solve.

For reference, table 9.4 shows at which point the time taken to compute the LUT is equal to the time taken to compute the score using the given $t$-size, to give an idea of the price of the overhead in practice.

9.6 Four-Russians and extended scoring system

We have seen that Four-Russian can be used to great effect as long as the scoring system can be encoded very tightly. For more practical scoring systems the encoding span a much bigger range of values, and as a result the size of the lookup tables quickly becomes too large.

<table>
<thead>
<tr>
<th>$t$</th>
<th>#tables</th>
<th>Table size (KB)</th>
<th>Total memory (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{1 \times 1}$</td>
<td>1 + 2</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>$t_{1 \times 2}$</td>
<td>1 + 4</td>
<td>128</td>
<td>640</td>
</tr>
<tr>
<td>$t_{2 \times 2}$</td>
<td>1 + 4</td>
<td>4,096</td>
<td>20,480</td>
</tr>
<tr>
<td>$t_{2 \times 3}$</td>
<td>1 + 6</td>
<td>131,072</td>
<td>917,504</td>
</tr>
<tr>
<td>$t_{3 \times 3}$</td>
<td>1 + 6</td>
<td>4,194,304</td>
<td>29,360,128</td>
</tr>
<tr>
<td>$t_{2 \times 4}$</td>
<td>1 + 8</td>
<td>4,194,304</td>
<td>37,748,736</td>
</tr>
<tr>
<td>$t_{3 \times 4}$</td>
<td>1 + 8</td>
<td>134,217,728</td>
<td>1,207,959,552</td>
</tr>
</tbody>
</table>

Figure 9.8: Memory usage for tables in extended local alignment with red highlighting for configurations that do not fit in memory on the test machine. Note that the number of tables needed is $-V_{min} + 1$, and $V_{min}$ is different for the extended scoring system.

Using the extended scoring system mentioned in the beginning of the chapter is a step towards a more realistic setting. As seen in table 9.8 some of the usual $t$-sizes have become unfeasible on the tested hardware, unless swapping is used. Using swap when memory runs out will definitely kill the performance, which is the sole purpose of using the Four-Russians technique in the first place. Only tables that fit in memory are considered for this test.
Figure 9.9: Extended scoring system. Table 9.8 shows the memory usage of each configuration.

Result

The result is seen in figure 9.9. There is still a speedup to be had, although the advantage of Four-Russians over the naive algorithm is diminishing. As expected the best $t$-block sizes have become even smaller, while the encoding has become more complex. Apart from a more complex encoding, more tables are also required to handle local alignment due to bigger interval of the extended scoring system.
Chapter 10

Backtracking

We have seen how Four-Russians can be used to speed up the scoring of two sequences, both for global and local alignment. The score itself is not particularly interesting. This chapter takes the final step of actually computing the alignment of a pair of sequences. Hirschberg’s algorithm is considered in the context of Four-Russians and local alignment.

10.1 Naive backtracking

If the full size \((n + 1) \times (m + 1)\) table is available, then backtracking is straightforward. The pseudo-code for backtracking in \(O(nm)\) space is given in figure 10.1. Note that bounds checks are left out.

```
while i > 0 or j > 0:
    if \(T_{i,j-1} + \text{gap} == T_{i,j}\):
        j--
        pair("-", \(Y_j\)) // Insertion
    i--
    pair(\(X_i\), "-") // Deletion
else:
    i--
    j--
    pair(\(X_i\), \(Y_i\)) // Match or mismatch
```

Figure 10.1: Pseudo-code for backtracking alignment in \(O(nm)\) space.

The starting position is assumed to be a part of the alignment. From there the algorithm backtracks by jumping to the neighbor from which it got its score. There can of course be several valid alignments when several neighbors contribute to a cell’s value. The backtracking part itself is only \(O(n+m)\) since we can at most decrement \(i\) \(n\) times and \(j\) \(m\) times. For global alignment \(i\) and \(j\) are set to \(n\) and \(m\), respectively, and the algorithm runs until we reach index \((0,0)\). Both sequences are completely included in the result, with gaps inserted. pair\((x, y)\) is the function which adds \(x\) to the beginning of the first aligned sequence and \(y\) to the other.

For local alignment we are only interested in the two subsequences of maximum similarity. They can of course still contain gaps in order to produce the best alignment according to the
scoring system. \( i \) and \( j \) are initially set to the maximum index found in the table. The algorithm does not terminate when \((0,0)\) is reached, but when the value of \( T_{i,j} \) reaches 0. See figure 6.1. The problem with the naive backtracking is the \( O(nm) \) space consumption. The absolute longest pair of input sequences is \( n = \sqrt{\frac{8 \text{GB}}{\text{sizeof(int)}}} - 1 \approx 46,340 \), with 8GB of RAM. This is without leaving space for the operating system and other parts of the algorithm. Since it is not possible to compute the alignment in a linear scan like the score, another approach is needed.

### 10.2 Hirschberg’s algorithm

Hirschberg’s algorithm [6] is able to compute the alignment in time \( O(nm) \) and \( O(\min(n,m)) \) space. The algorithm builds on two principles. One, that it is possible to compute a given row or column in an alignment problem in linear space, as seen in equation 2.4. Two, index \( T_{i,j} \) will contain the score for the global alignment of \( S_1[0..i] \) and \( S_2[0..j] \). When computing the table starting at the lower-right corner, i.e. the reverse sequences, index \((i,j)\) will contain the optimal alignment score of \( S_1[i..n] \) and \( S_2[j..m] \).

Hirschberg’s algorithm was originally designed to compute the longest common subsequence in linear space, but is often and easily adapted to sequence alignment. The second principle is the clever bit. Consider the input sequences

\[
S_1 = \text{ttccgct} \quad \text{and} \quad S_2 = \text{ccgtgct}
\]

Computing unit cost edit distance we get the following tables for the alignment of \((S_1, S_2)\) and \((\text{reverse}(S_1), \text{reverse}(S_2))\). Note that in the illustration below \( T^r \), the table for the reversed sequences, is visualized by starting the computation at the lower-right corner of the table, which is equivalent to computing the reversed input sequences and flipping the resulting table in both directions.

<table>
<thead>
<tr>
<th>( T )</th>
<th>( T^r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>c</td>
<td>4 4 3 3 4 5 6 7 t</td>
</tr>
<tr>
<td>g</td>
<td>t 1 1 2 3 3 4 5 6</td>
</tr>
<tr>
<td>t</td>
<td>3 3 3 2 3 4 5 6 t</td>
</tr>
<tr>
<td>c</td>
<td>t 2 2 2 2 2 3 4 5 c</td>
</tr>
<tr>
<td>c</td>
<td>2 2 2 1 1 2 3 4 c</td>
</tr>
<tr>
<td>g</td>
<td>c 4 3 2 1 0 1 2 3 g</td>
</tr>
<tr>
<td>c</td>
<td>3 2 2 1 0 1 2 3 c</td>
</tr>
<tr>
<td>g</td>
<td>4 3 2 1 0 1 2 3 g</td>
</tr>
<tr>
<td>t</td>
<td>5 4 3 2 1 0 1 2 3 t</td>
</tr>
<tr>
<td>c</td>
<td>6 5 4 3 2 1 0 1 2 t</td>
</tr>
<tr>
<td>g</td>
<td>7 6 5 4 3 2 1 0 1 c</td>
</tr>
<tr>
<td>c</td>
<td>8 7 6 5 4 3 2 1 0 c</td>
</tr>
</tbody>
</table>

Figure 10.2: Computing the unit cost edit distance of a pair of sequences in forward and reverse order.

For any index \((i,j)\), \( T_{i,j} + T^r_{i,j} \) is the score of the alignment including the cell at \((i,j)\). This means that any \((i,j)\) where \( T_{i,j} + T^r_{i,j} \) equals the optimal score is a part of a valid optimal alignment. Element-wise addition of \( T \) and \( T^r \) will yield a table where it is easy to spot several indices (with value 4) which are part of an optimal alignment. See figure 10.3.

The idea is to use divide-and-conquer and treat the alignment as a bunch of subproblems without knowledge of the full table, which would require \( O(nm) \) space. The first step is to pick the middle row in the current subproblem, \( s = n/2 \). The goal is then to find the \( s \)th row in
\[
T + T^r = \\
\begin{array}{cccccccc}
4 & 5 & 5 & 6 & 8 & 10 & 12 & 14 \\
4 & 4 & 5 & 5 & 6 & 8 & 10 & 12 \\
4 & 4 & 4 & 5 & 5 & 7 & 9 & 10 \\
6 & 4 & 4 & 4 & 5 & 6 & 7 & 9 \\
8 & 6 & 4 & 4 & 4 & 6 & 6 & 8 \\
10 & 8 & 6 & 4 & 4 & 6 & 6 & 7 \\
12 & 10 & 8 & 6 & 5 & 5 & 4 & 6 \\
14 & 12 & 10 & 8 & 6 & 6 & 6 & 4 \\
\end{array}
\]

Figure 10.3: Element-wise addition of \(T\) and \(T^r\) shows which cells are part of an optimal alignment.

\(T + T^r\) in linear space, let us call them \(L\) and \(L^r\), respectively. Since it is possible to compute any row in an alignment table in linear space, these rows can also be computed in linear space. \(L\) is the last row in the alignment of \(S_1[0..s]\) and \(S_2[0..m]\). \(L^r\) is aligned using the reverse sequences on the "bottom" part of the subtable, \(S_1^r[0..s]\) and \(S_2^r[0..m]\). Using element-wise addition of \(L\) and \(L^r\) we get the \(s\)th row of \(T + T^r\).

\[
L + L^r = \\
\begin{array}{cccccccc}
6 & 4 & 4 & 4 & 5 & 6 & 7 & 9 \\
\end{array}
\]

The optimal score might not be known in advance. For instance in global alignment, there is no point in scoring the entire problem before running backtracking. Also, the optimal scores in the subproblems are not the same as the main problem. Instead of looking for, in this case the value four, we can simply find the best value in \(L + L^r\) at index \(k\). Several best \(k\)s may exist, but they all form an optimal alignment in the end. At this point we know that cell \((s,k)\), in the current subproblem, is part of the final optimal alignment. This is where the algorithm recurses in order to find the subalignment before and after \((s,k)\). The prefix is found by aligning \(S_1[0..s]\) against \(S_2[0..k]\) and the suffix is found by aligning \(S_1[s..n]\) against \(S_2[k..m]\) recursively. See figure 10.4.

Since the crossing point \((s,k)\) is known, the two subproblems can be seen as completely independent alignments and there is no need to propagate the input edges to the them. The subproblems are simply initialized like the outer problem, with \(T_{i,0} = i \cdot \text{gap}\) and \(T_{0,j} = j \cdot \text{gap}\), and the subsequences are used as input sequences.

### 10.2.1 Base case

The base case is reached when \(n = 1\) and the subproblem cannot be divided further. \(m\) may still be \(\geq 0\). The correct crossing point, \(k\), and the whether it is a diagonal or vertical cross, must be found.

Figure 10.5 shows the two base cases. The \(m\) possible diagonals are iterated to find the best match of \(S_1[0]\) and \(S_2[k]\). There is only one crossing since \(n = 1\), and gaps must be inserted on, potentially, either side of that crossing.

The second case is a vertical gap. There is no need to iterate all the possible crossings since this case does not depend on the input sequences, and is always \(m \cdot \text{gap} + \text{gap}\). In a scoring system where \(\text{cmp}_{\text{min}} \geq 2 \cdot \text{gap}\), this case is only ever taken when \(m = 0\) since no diagonal crossing exists, and a diagonal crossing would otherwise always be preferred.
10.2. HIRSCHBERG’S ALGORITHM

Figure 10.4: Once the crossing point, \((s,k)\), is found the alignment can be recursively created by concatenating the subalignment found in the blue region with the subalignment in the green region.

\[ m = 3 \]
\[ S_2[0] \quad S_2[1] \quad S_2[2] \]
\[ n = 1 \quad S_1[0] \]

Figure 10.5: The two base cases for Hirschberg’s algorithm. Left: Diagonal crossing with 0 or more gaps. Right: Horizontal crossing.

\[ gap \cdot (m - 1) + \max_k (\text{score}(S_1[0], S_2[k])) \]
\[ gap \cdot m + gap \]

10.2.2 Time complexity

\( s \) rows are computed in order to find \( L \) and \( n - s \) rows for finding \( L' \). These rows are \( m \) long. The first step thus takes time \( s \cdot m + (n - s) \cdot m = n \cdot m \) steps. When the algorithm recurses, each sub problem has roughly height \( n/2 \). The width for the prefix is \( k \) and the suffix is \( m - k \). The time taken for the second level of the recursion is \( k \cdot (n/2) + (m - k)(n/2) = m \cdot (n/2) \). The third level is \( m \cdot (n/4) \) and so on, which adds up to \( 2mn \), or \( O(nm) \) time.

This is the same time bound as the naïve algorithm, but space is reduced by a factor \( O(\max(n,m)) \).

10.2.3 Local alignment

The algorithm given in section 10.1 stops when a 0 is encountered in local alignment. Fortunately, using a first pass for finding the local region, Hirschberg’s can be made to work on local alignment as well.
Finding the end point of the local region, \((h,l)\), is easily done using the existing implementation. However, local alignment backtracking should stop when a 0 is found so running Hirschberg’s in region \((0,0),(h,l)\) will not produce the correct result. The idea is then to find the end point on the reversed input sequences, which would correspond to the starting point in the original pair of sequences. There is one catch however.

![Figure 10.6: Finding the optimal region in a local alignment problem](image)

Consider the left part of figure 10.6 which has two optimal alignments, \(t\) and \(t'\). The forward pass finds the endpoint, \((h,l)\), of \(t\). However, since the maximum value may occur more than one place in the table several alignments may therefore exist, even completely independently. The reverse pass might not find the start coordinates that correspond to the end found in the forward pass. In practice this works most of the time, but not always.

A more correct solution is to run the reverse pass on sequences \(\text{reverse}(S_1[0..h])\) and \(\text{reverse}(S_2[0..l])\), as seen in the figure to the right. This forces the algorithm to find the maximum value ending at \((h,l)\), and also saves some computation if \((h,l)\) is much smaller than \((n,m)\).

### 10.2.4 Zero-case

Once the start and end coordinates are found for the local alignment it can be treated as a global alignment problem of \(S_1[i..h]\) and \(S_2[j..l]\). The zero-case in the local recursion should be ignored when Hirschberg’s algorithm is run. The local area has already been determined, and the global alignment of that area must be found.

### 10.3 Four-Russians and Hirschberg’s algorithm

Finding \(L\) and \(L'\) makes up most of the work in Hirschberg’s algorithm. The Four-Russians method can be used to speed this step up. Computing a number of lines which is not evenly divisible by the \(t\)-block size, should be handled somehow. Kundeti and Rajasekaran [7] suggest that the last row can be slid back into the second to last row when \(n\) is not a multiple of \(t_n\). This does mean keeping track of extra information in order to transform \(B\) and \(C\) to the correct values for the last block, and will also require an edge case when \(n < t_n\) and there is nothing to slide back into. They also suggest the possibility of simply not using Four-Russians for the remaining
\( n \mod t_n \) lines. Since the encoded edges used in Four-Russians must be converted back into an array of values in order to determine the crossing point, there is no overhead associated with computing \( n \mod t_n \) lines using the normal method, other than the computation itself. (As seen in table 10.1 the overhead of computing the \((n \mod t_n) \cdot m\) cells naively is barely noticeable.)

Hirschberg’s will find a global alignment, which means the zero-case in the recursion is no longer relevant. The scoring system is still the same. In case of Four-Russians we can simply just use the lookup table for \( A \geq -V_{\min} \), which of course is equivalent to using a table where the zero-case is not present.

It was determined earlier that the Four-Russians method, with the layout of \( YBXC \), works best in column-order when \( Y \) could be fixed for a long time. To accommodate this, the implementation of Hirschberg’s algorithm works transposed to what was described in section 10.2, such that the crossing point of the middle column is found in each recursive step.
10.4 Experiments

The result is easily verified for correctness by summing each pair in the alignment against the scoring system. If the accumulated sum matches the score, the alignment is indeed optimal. The verifying code also considers the start index of the alignment and verifies that the symbols in the alignment corresponds to the ones in the input sequences in correct order.

10.4.1 Naive vs hirschberg’s algorithm

Both the naive backtracking (including table computation) and Hirschberg’s algorithm runs in theoretical $O(n^2)$ time. Running both implementations against varying input sizes and plotting the result divided by $n^2$ we should, in theory, see a perfectly straight line.

![Graph showing time vs sequence length for naive backtracking and Hirschberg’s algorithm](image)

**Figure 10.7:** Verifying that the naive backtracking and Hirschberg’s algorithm stay within the theoretical $O(n^2)$ time bound.

Result

See figure 10.7. As expected both algorithms follow the $O(n^2)$ time bound nicely. There is a bit of fuzz at $n < 1000$, but at greater input both algorithms converges. Longer test sequences become a problem on the test hardware when using the naive algorithm, due to insufficient memory.

Even though Hirschberg’s algorithm has some constant hidden in the big-O notation, it is still faster than the naive approach for most input sizes.

10.4.2 Hirschberg’s and Four-Russians

In chapter 5 we saw that edit distance calculation could be speed up by an order of magnitude with some choices of $t$ and input size. This test seeks to find out if Hirschberg’s algorithm for edit script can also be sped up. In this test a global alignment is found in order to ensure that Hirschberg’s algorithm runs on the full $n \times m$ problem for a given input size.
10.4. EXPERIMENTS

10.4.1 Sequence length \((n\text{ and } m)\)

\[
\begin{align*}
\text{Speedup ratio vs naive} \\
\text{Figure 10.8: Different sizes of } t \text{ for Hirschberg’s algorithm and Four-Russians in finding edit script.}
\end{align*}
\]

**Result**

See figure 10.8. Compared to edit distance computation in figure 5.5 the speedup for edit script is not quite as impressive. Some configurations, \(t_{1\times1}\), \(t_{3\times3}\) and \(t_{3\times4}\), are even slower than normal, non-Four-Russians for many of the tested input sizes, likely because the middle row has to be converted back to an array of values instead of the Four-Russians encoding when the middle column has to be found. The entire computation of edit distance can be kept in the Four-Russians encoding and there is not a lot of overhead. The characteristics of different \(t\)-block sizes remain. \(t_{2\times2}\) and \(t_{2\times3}\) seems to be the obvious choices, while \(t_{2\times4}\) starts to become relevant for large input.

10.4.3 Subproblem handling: Naive vs Hirschberg’s algorithm

Each recursive function call in Hirschberg’s algorithm must have \(L\) and \(L^r\) available in order to determine the optimal crossing point. The Four-Russians variant must therefore convert the encoded edges back in order to compute the crossing point \(k\). This adds some overhead and in case of the smallest subproblems it might be better to simply call the naive \(O(nm)\) space backtracking algorithm when \(n\) or \(m\) gets below a certain value. What this value is, is tested in this experiment.
CHAPTER 10. BACKTRACKING

10.4.4 Time distribution in local alignment

Using high precision timers the time taken for different parts of the Four-Russians variant of Hirschberg’s algorithm can be accumulated. In table 10.1 the different tasks, and their accumulated execution times, are listed. The "Other implementation overhead" field was computed in separate run without using the timers where the total time was subtracted by all the timers. The default scoring system from chapter 9 was used.

Result

Time distribution for the naive $O(nm)$-space algorithm is not very interesting. When aligning a pair of length 35,000, 8.68 seconds are spent computing every entry in the table and noting the entry with the best score. The actual backtracking part only takes 2.2 milliseconds. This time distribution is somewhat expected since the backtracking part runs in time $O(n + m)$.

Different things can be concluded from the data from local alignment backtracking in table 10.1:

1. The vast majority of time is spent finding the local region of interest and computing $L$ and $L'$ inside Hirschberg’s. It is to be expected that both tasks use roughly the same amount of

Figure 10.9: Left: Backtracking time for the naive $O(nm)$ algorithm (Red), and Hirschberg’s using Four-Russians with $t = (2,3)$ (Green) given different combinations of input size $n$ and $m$. Right: Any rise over 0 is where the naive algorithm outperforms Hirschberg’s Four-Russians, for a given $n \times m$ size.

Result

See figure 10.9. For the optimal $t$-size of $(2,3)$ Hirschberg’s algorithm, combined with the Four-Russians, is always faster except for sizes where either $n$ or $m$ is 1 and the other input sequence is longer. The advantage of the naive algorithm grows linearly when $n = 1$ and $m$ becomes larger. This is not the case for $n$ and $m$ swapped. This is likely due the the column orientation which will recurse the same number of times independently from $n$. In practice is it unlikely that the algorithm will encounter such a case for a large value of $m$ since this will correspond to a long gap in input $S_1$. Also, a simple extra check in the code could handle this case efficiently. This rules out the viability of a variant which will revert back to the $O(nm)$ space algorithm for sufficiently small subproblems.
10.4. EXPERIMENTS

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (s)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local area (+LUTS)</td>
<td>21.2503</td>
<td>54.329%</td>
</tr>
<tr>
<td>Finding $L$ and $L^r$ (Four-Russians)</td>
<td>17.7876</td>
<td>45.476%</td>
</tr>
<tr>
<td>Other implementation overhead</td>
<td>0.0457</td>
<td>0.117%</td>
</tr>
<tr>
<td>$n \mod t$ naive compute</td>
<td>0.0122</td>
<td>0.031%</td>
</tr>
<tr>
<td>Encoded edges to values</td>
<td>0.0089</td>
<td>0.023%</td>
</tr>
<tr>
<td>LUT for backtracking</td>
<td>0.0062</td>
<td>0.016%</td>
</tr>
<tr>
<td>Base case</td>
<td>0.0033</td>
<td>0.008%</td>
</tr>
</tbody>
</table>

Table 10.1: Time distribution between tasks when backtracking two sequences of length 100,000 using the Four-Russians variant of Hirschberg’s algorithm with a $t$-block size of (2,3).

time. The local region is almost the same size as the entire global problem for this scoring system, so the lower-right corner of the region, $(h,l)$, is found by noting the best score in $(0,0), (n,m)$. And the upper-left is found by scoring $(h,l), (0,0)$, the reverse sequences. This is around $2n^2$ computations, which is also roughly what Hirschberg’s will use. Hirschberg’s only requires one lookup-table, whereas the local area computation requires $-V_{\text{min}} + 1$, which is likely why it is faster even considering the extra overhead of divide-and-conquer.

2. The base case itself is a microscopic part of the total computation when the input is somewhat large. Even if the algorithm were to fall back to naive computation earlier in the recursion, it would still be a small part of the computation. The accumulated value of $k$ in all the base cases is $m$, because all the base cases combined must span the width of the table. The base case is invoked $n-1$ times during the execution, so handling the base case itself is a $O(n + m)$ operation.

3. How the odd edges are handled does not really matter. Sliding the $t$-blocks back in order to avoid naive computation at all does not seem worth the effort when there is only potential to improve 0.031% of the execution time. Also, the overhead of converting back and fourth between arrays of values and Four-Russians encoding is barely noticeable.

4. As we have already established in previous chapters, the computation of a medium sized $t$-block table is very small part of the total computation. The LUT construction overhead does not really matter, except on small input.

10.4.5 Time distribution vs input similarity

In case of local alignment, input similarity has an effect on the execution time. The backtracking part itself only requires a single table. However, the first pass in finding the local area is computed with the usual approach, using several tables. How expensive it is to find the start point of the area also depends on the end point found beforehand. The location of the points very much depend on the input similarity.

Result

The search for the end point is the first step. As seen in figure 9.6 local alignment Four-Russians is fastest at low or high similarity. Unsurprisingly, the same is true here, see figure 10.10. Afterwards the reverse search is run using the end point as bound. For low similarity, the similar area is very small and the second computation, for finding the starting point, is bounded to a
very small area. Thus the cost for this step is much less than $n \cdot m$ the lower the input similarity gets. Both search directions approach $n \cdot m$ cost when the problem essentially becomes global. The same is true for the actual computation of Hirschberg’s.

### 10.4.6 Extended scoring system

In practice a more detailed scoring matrix is often needed. This will result in more combinations and thus a larger lookup-table is required. The extended scoring system, used in chapter 9, is tested where three bits are used for each jump, as opposed to two. For local alignment this means that each lookup table will have $32^{t_n+t_m}$ entries.

**Result**

This result, in figure 10.11, is similar to the results obtained when computing the extended score of an alignment. It is still feasible to use Four-Russians here, but the gains are diminishing.

The $t_{3 \times 4}$ and $t_{2 \times 4}$-tables are no longer possible with 8GB of RAM. A single $t_{2 \times 4}$-table, using this scoring system, requires $(I|\Sigma|)^{2+4} \cdot \text{sizeof(entry)} = 4$GB of RAM. Using equation 6.4
Figure 10.11: Finding the local alignment of a pair of sequences using Four-Russians, Hirschberg’s algorithm and the extended scoring system. The time includes finding the local region and backtracking. 

the number of distinct tables required in the first pass, when finding the local region, is 9. $9 \cdot 4\text{GB} = 36\text{GB}$ of RAM. Attempting $t_{3\times 4}$ or bigger is unrealistic, even on future hardware.
Chapter 11

Conclusion

The Four-Russians technique is very much applicable for certain local alignment problems. However, the size of the speedup gains over the naive dynamic programming algorithm is inversely proportional to the number of bits needed to encode the scoring system. We have seen that the best block-size, and Four-Russians’ advantage, quickly shrinks when more advanced scoring systems are utilized. Despite this, there is often a several factor speedup to found using the ideas presented in chapter 6. The algorithm seems to work best when the scoring system is fairly compact. A bigger range of weights result in a more complex encoding and also more tables required for local alignment.

The Four-Russians method was developed in a time when lookup tables were the solution to all performance related problems, and while CPUs have followed Moore’s law closely up until today, the same is not quite true for the speed and latency of memory. It turns out that the performance of Four-Russians is not limited by the amount of available memory, but the CPU cache’s size and speed. The key factors to Four-Russians’ performance are summarized below:

- **t-block shape and lookup table layout.** Depending on whether the computation is row- or column-major, the best shape seems to be flat perpendicular to the direction of computation as seen in figure 5.3. The layout of the encoding is also extremely important when trying to utilize caching best.

- **The size of \( t_n + t_m \).** For different problems and normal input sizes the best \( t \)-blocks, and their total memory usage, are listed below:

  - **Global, unit cost:** \( t_{2 \times 4} - 64\text{MB} \) (\( \approx 11\text{MB} \) without waste)
  - **Local:** \( t_{2 \times 3} - 16\text{MB} \)
  - **Local, extended:** \( t_{2 \times 2} - 12\text{MB} \)

  They all share roughly the same memory footprint. Of course, the access pattern is restrict to a smaller area when run, as described in section 5.3.

- **Input size, specifically \( n \).** When \( n \) grows extremely large, the results change a bit when the problem essentially becomes \( m \) separate computations of size \( n \). This is of course assuming a column-major computation direction.

The behavior of the cache also plays a huge role in which combination of input- and block-size gives the best result. We have seen that medium-sized blocks perform well for most input sizes, but the truly big blocks only begin to make sense when the input grows to extreme sizes.
11.1. FUTURE WORK

It is indeed also possible to improve the Four-Russians preprocessing step quite a bit with a new algorithm. While the improvement is nice, it is simply not worth it in combination with the actual Four-Russians computation. The smaller LUTs, that are best for small input sizes, are already constructed extremely quickly using the naive algorithm. The naive construction takes roughly five seconds to generate the tables needed for $t_{2\times4}$ local alignment. However, this block-size only becomes optimal when the Four-Russians runs for several hours. Despite all the tricks used to improve the dynamic programming approach used for sequence alignment, it is still a $O(n^2)$ algorithm underneath. Therefore, the preprocessing time has little relevance in the grand scheme of things.

11.1 Future work

Listed below are some ideas that could be interesting to explore given more time.

- When handling the zero-case in local alignment, the idea has been to do something else about blocks where a low value of $A$ might cause problems. $A$ taking a dangerous value does not necessarily mean that the block will hit the zero-case. In case both $B$ and $C$ are rising in value, for instance $[2, 2, 2]$, it is unlikely that any cell in the block will reach a value below $A$. This means that there is some redundancy in the tables generated for the same indices, but different values of $A$.

A way to potentially reduce the number of tables needed, is to instead define $V_{\text{min}}$ as the lowest internal cell relative to the lowest cell in the input, $A$, $B$ or $C$. The invariant is that the input cells can never reach a value below zero. The problem is that the implementation will have to determine the value of the lowest point in the index before looking up the correct block. This could be done using a small lookup-table, like the edge-sum LUT in section 3.4.3 or it could be encoded for each block’s $B'$ and $C'$ output. The layout of the table will also have to be modified such that it is possible to lookup a block, with encoding $YBXC$, but a certain value of $V_{\text{min}}$.

This idea would likely reduce redundancy and waste, which is important as seen in section 5.6. However, the additional overhead might be a problem.

- Attempting to reduce the memory usage in local alignment by naively computing some of the less used tables was fairly unsuccessful. Maybe a smarter scheme could be created by looking at the input’s similarity and analyzing which tables would be used most from that. It might also make sense to use a smaller $t$-block size as fallback instead of the naive algorithm.

- This thesis has only considered constant gap penalties. Smith-Waterman is also able to handle affine gap costs. Adapting Four-Russians to affine gaps might be interesting.

- Finally, graphics processing units (GPUs) can be more than an order of magnitude faster traditional CPUs for parallel workloads. It is certainly possible to compute the dynamic programming table in a way such that a range of cells are computed independently in parallel. It might be interesting to see how a GPU’s memory model, which is more geared towards throughput than latency, fits with Four-Russians. The advanced fast caches in the CPU was also shown to be a big factor in the performance. GPUs have limited, simpler caches, but more raw processing power to compensate.
Bibliography


[4] Espeholt, Lasse. 2013. Exploring the practicality of the Four Russians method for sequence alignment. [https://docs.google.com/file/d/0B1Z0_s5g90zHWEtiNXpMV0ZUUEE](https://docs.google.com/file/d/0B1Z0_s5g90zHWEtiNXpMV0ZUUEE)

