Department of Computer Science
Aarhus University
Denmark

Speeding-up dynamic programming in sequence alignment

Master's Thesis

Dung My Hoa - 20022443

December 1, 2010

Supervisor: Christian Nørgaard Storm Pedersen

Implementation code : http://www.daimi.au.dk/~dmh/thesis/dmh20022443_thesis.zip

Abstract

Computing the optimal cost and alignment of two sequences is one of the most fundamental problems in bioinformatics. The standard dynamic programming based algorithm computes the optimal cost and alignment in $O\left(n^2\right)$ time and space. Hirschberg gave an algorithm to compute the optimal alignment in $O\left(n\right)$ space, but the time remained $O\left(n^2\right)$. The first major improvement of the asymptotic running time was with the Four-Russians speedup. It reduced the time to $O\left(\frac{n^2}{\log n}\right)$ for computing the optimal cost, but it did not address how to compute the optimal alignment within that time. That was given by Kundeti and Rajaskaran, who combined Hirschberg's algorithm with the Four-Russians speedup. However the Four-Russians speedup is not commonly used in practice, this is perhaps due to the overhead of using the Four-Russians speedup.

The motivation of this thesis is to investigate if there is a practical speedup using Four-Russians on dynamic programming in sequence alignment. This thesis also discusses issues which arise when implementing the Four-Russians speedup for edit distance and script in quadratic as well as linear space. Also, implementations using the Four-Russians with a theoretical speedup of $O\left(t\right)$ rather than $O\left(\log n\right)$ are presented together with experiments showing the performance of these, and an evaluation of the Four-Russians speedup applicability on dynamic programming in sequence alignment.

Contents

1 Introduction

One of the most common methods used for inferring the biological function of genes is sequence similarity search in protein and DNA sequence databases. With the development of rapid methods for sequence alignment, results based solely on sequence homology have become routine. Although dynamic programming based sequence alignment does provide optimal solutions they are computationally expensive. Therefore, the most commonly used methods are currently based on heuristics which are much faster, such as BLAST (Basic Local Alignment Search Tool)[1], at the cost of providing optimal solutions. Speed is important given the size and growth of the sequence databases currently available. So the continuing development of fast and accurate algorithms is appealing.

The first algorithm introduced using dynamic programming in global alignment, was by Needleman and Wunsch [13]. Later algorithms for variations of global alignment, such as local alignment and affine gap cost, were developed in [16], [6] and [10]. The first major improvement in the asymptotic running time was achieved in [1], also known as the Four-Russians speedup or Four Russian algorithm[2]. The algorithm improves the running time for computing the optimal cost by a factor of $O(\log n)$ but it did not address how to compute the optimal alignment within the same running time. Hirschberg [9] gave an algorithm for computing the optimal alignment in $O(n^2)$ time and $O(n)$ space. The space saving idea in Hirschberg's algorithm was applied in [7] and [12]. However the asymptotic running time of computing the optimal alignment remained the same $O(n^2)$. Also, parallel algorithms for the optimal cost were studied in [3] and [15]. In [14] linear space parallel algorithms were given, however the asymptotic running time was still assumed to be $O(n^2)$. A survey on all these algorithms can be found in [8]. In [11] Hirschberg's algorithm were combined with the Four-Russians speedup giving an $O\left(\frac{n^2}{\log n}\right)$ algorithm for computing both the optimal cost and alignment in $O(n)$ space.

1.1 Motivation

Sequence alignment is a fundamental application in bioinformatics where it used to infer functional, structural and evolutionary relationships between sequences. This makes it an interesting area to explore speedup possibilities. Even though the Four Russian paradigm have been applied for a number of dynamic programming algorithms an actual implementation is rare [4], [5]. The motivation of this thesis is to explore practical speedup possibilities of dynamic programming in sequence alignment. I decided to focus on the Four-Russians speedup as the theoretical speedup is a major improvement in the computation of sequence alignment. I will investigate the Four-Russians speedup applicability on dynamic programming in global sequence alignment and then compare the Four-Russians speedup with a standard dynamic programming based algorithm [13] and Hirschberg's algorithm [9].

1.2 Objectives

The main objective of this thesis is to evaluate the applicability of Four-Russians speedup on dynamic programming in sequence alignment. I also aim to show that the Four-Russians speedup

[1] http://en.wikipedia.org/wiki/BLAST
[2] Although only one of them was Russian.

does not just give a theoretical but also a practical speedup in the computation of sequence alignment. The theoretical running time can not always be achieved in practice, as the practical running time is dependent on the implementation choices. To fully understand the practical running time, a discussion on the implementation issues and details which follows an implementation of the Four-Russians speedup and how they may affect the practical running time is presented. There are several elements that need to be taken into account. How to handle data, how to speedup the preprocessing part and how to efficiently applied the Four-Russians speedup on a standard implementation of dynamic programming in sequence alignment. The implementations will be based on edit distance and script for global alignment in quadratic as well as linear space.

1.3 Thesis outline

Section 2 will give an introduction to sequence alignment and the algorithms used in the different implementations. Section 3 presents implementation issues and details on procedures and elements which are common for both the quadratic and linear space implementation of the Four-Russian speedup. Experiments used to test the implementations and the expectations of these are also presented in section 3. In section 4, 5, 6 and 7, the implementation issues and details for the different implementations will be presented followed by the results of the experiments. To summarize the experimental results for the different implementations, an overview and a discussion comparing the results will be presented in section 8. Other results which are not presented in section 4, 5, 6 and 7 will also be presented here together with a discussion of these. A discussion on other speedup possibilities and some ideas to further improve the practical running time of the Four-Russians speedup in sequence alignment will be presented in section 9. Finally in section 10 the applicability of the Four-Russian speedup on dynamic programming in sequence alignment will be evaluated.

2 Background

In this section I will give an overview of the algorithms used for the different implementations. A detailed introduction to sequence alignment, dynamic programming and Four-Russians speedup. I will also to cover the terminologies, notations and ideas used for the different implementations. Hirschberg's algorithm is only used to compute an optimal alignment in linear space, so it is not covered in this section, but in section 7 along with the implementation issues and details. In this section the sequences are assumed to be of equal length. How to handle sequences of different length will be discussed in the implementations details for the different implementations.

2.1 Sequence alignment

In bioinformatics, a sequence alignment is a way of arranging DNA, RNA, or protein sequences to identify regions of similarity. Highly similar regions may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns[3].

[3]http://en.wikipedia.org/wiki/Sequence_alignment

2.1.1 Interpretation

Two aligned characters corresponds to either a match in both sequences or a substitution from sequence A to sequence B, i.e point mutations[4]. A gap introduced in sequence A corresponds to a insertion in sequence B, and a gap introduced in sequence B corresponds a deletion in sequence A, i.e indels[5].

2.1.2 Optimal cost and alignment

The similarity of two sequences is measured as a cost for transforming sequence A into sequence B. There are different types of cost measures. A commonly used measure, the edit distance, counts the number of operations required for the transformation. In a generalization of the edit distance, the cost of each operation is summed instead of just the number of operations needed to transform one sequence into another. Sequence alignment is about optimizing the cost of the alignment. To emphasize similarity the objective is to maximize the number of matches. To explaining differences the objective is to minimize the number of indels and point mutations, hence this is an optimization problem.

There are different types of edit distances such as the Hamming distance[6], which only measures substitutions between sequences. The Levenshtein distance[7], which measures all the operations mentioned in section 2.1.1, normally referred to as "edit distance". And the Damerau-Levenshtein distance[8], which includes another operation which is not mentioned in section 2.1.1, a transpose of two adjacent characters.

Computing the edit distance corresponds to setting the cost of each operation to one. In the general case each operation can have different cost which means that substitutions, insertions and deletions can be weighted differently. To represent the cost of different operations a cost table and a gap function is typically used.

An optimal alignment is an alignment with an optimal cost. Regions of mutations can be identified from an optimal alignment, so often the optimal alignment is more important than the optimal cost. Henceforth edit distance will refer to the Levenshtein distance, and edit script will refer to an optimal alignment with the optimal edit distance.

2.1.3 Time and space

A naive algorithm to compute the edit distance is shown in figure 1(a). Implementing this naively without storing edit distances to sub-sequences is very time consuming, as the edit distance for sub-sequences will be computed multiple times. As for the space it only depends on the size of the sequences, i.e. $O(n)$.

Computing an edit script is almost the same as computing the edit distance. The only difference is that after the optimal edit distance have been found for an entry, we need to backtrack, i.e. compute the alignment which gave rise to the edit distance. An algorithm for computing the edit script is shown in figure 1(b). Finding the edit script is even more time consuming than computing

[4]http://en.wikipedia.org/wiki/Point_mutation
[5]http://en.wikipedia.org/wiki/Indel
[6]http://en.wikipedia.org/wiki/Hamming_distance
[7]http://en.wikipedia.org/wiki/Levenshtein_distance
[8]http://en.wikipedia.org/wiki/Damerau-Levenshtein_distance

```
optCost(i,j):
  d, v, h, s = undef
  if i > 0 and j > 0
    d = cost(i-1,j-1) + ed(A[i],B[j])

  if i > 0 and j ≥ 0
    v = cost(i,j-1) + 1

  if i ≥ 0 and j > 0
    h = cost(i-1,j) + 1

  if i = 0 and j = 0
    s = 0
  return min(d,v,h,s)
```

(a) Computing the edit distance naively.

```
optAlign(i,j):
  same as in cost function except the last line
  o = min(d,v,h,s)
  if o = d
    optAlign(i-1,j-1)
    align A[i] with B[j]
  if o = v
    optAlign(i-1,j)
    align A[i] with a gap
  if o = h
    optAlign(i,j-1)
    align B[j] with a gap
  if o = s
    return
```

(b) Computing the edit script naively.

Figure 1: Simple implementation.

the optimal edit distance because of additional calls to the cost function. Space consumption remains the same as computing the edit distance.

2.2 Dynamic programming

Dynamic programming is a method of breaking complex optimization problems into smaller optimization sub-problems, which can be combined to solve entire optimization problems. To be able to use dynamic programming the problem has to consist of slightly smaller overlapping sub-problems. Also the procedure to solve the problem must be to repeatedly solve the same sub-problem. The idea is to store solutions to sub-problems, which can be retrieved later to solve bigger sub-problems and thereby solve the entire optimization problem[9].

Sequence alignment is an optimization problem, where the optimal solution can be computed from combining the optimal solutions from slightly smaller sub-problems. The solutions to the sub-problems are used multiple times to compute other sub-problems, i.e. overlapping sub-problems. So applying dynamic programming to sequence alignment will improve the computation time of finding the edit distance and script. The edit distance for each pair of sub-sequences are stored in a distance table, so they can be retrieved for later use.

2.2.1 Time and space

An algorithm for computing the edit distance with dynamic programming is almost the same as the naive version. Iterate over $i = 0, \ldots, n$ and $j = 0, \ldots, n$ and replace each recursive function call with a lookup in the distance table. This way each value that is required to compute the edit distance in an entry is already computed and can be retrieved from the distance table. This is also known as forward dynamic programming. The time complexity is $O\left(n^2\right)$, as each entry in the distance table is computed once in constant time. There are $(n+1)^2$ pairs of sub-sequences as the empty sequence is also treated as a sub-sequence, hence the space consumption is $O\left(n^2\right)$.

When the distance table has been filled out, computing the optimal edit script can be done by backtracking in the distance table. The algorithm is almost the same as the naive version, but with

[9]http://en.wikipedia.org/wiki/Dynamic_programming

each recursive function call replaced with a lookup in the distance table. Making three lookups in the distance table for each entry takes constant time, which means the time consumption only depends on the number of entries we need to visit in the distance table. Worst case is when sequence A and B is only aligned with gaps, hence $O(n)$ time.

2.3 Four-Russians speedup

Four-Russians speedup is a method to speedup dynamic programming. The general idea of Four-Russians speedup is to partition the distance table into t-blocks and compute essential values in the table one block at a time. The essential values in the distance table are the values required to compute a block. The goal is to only use $O(t)$ time on each block, instead of the normal $\Theta(t^2)$ [8].

2.3.1 t-block

Now consider the standard dynamic programming procedure of computing the edit distance in a block in the distance table (see figure 2). The block D is computed from the sub-seq A and B, start value S, row R and column C. It is clear that the block D is a function of these, hence the last row and column of the block is a function of sub-seq A and B, start value S, row R and column C.

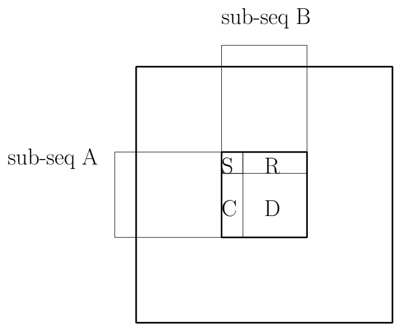


Figure 2: A block in the distance table.

Let a t-block be a block of size t in the distance table, where the last row in the block is shared with the first row in the block below it (if any), and the last column in the block is shared with the first column in the block to its right (if any).

2.3.2 Block function

Given sub-seq A and B, start value S, row R and column C the block function computes the last row and column of the block. The computation time is $\Theta(t^2)$ when done naively. The goal is to only use $O(t)$ on each block. One way is to precompute all possible inputs for the block function, so that the last row and column can be computed in $O(t)$ time. By definition each entry can hold a distance value from zero to n, so there are $n+1$ possible values for any t-length row and column. Hence the possible input combinations to the block function is $(n+1)^{2t}\sigma^{2t}$, where σ is the size of the alphabet. For each input, the block function takes $\Theta(t^2)$ time to compute the last row and column of a block. So the overall time to precompute the function output is $\Theta((n+1)^{2t}\sigma^{2t}t^2)$. But as t is at least one this gives a $\Omega(n^2)$ precomputation time. So there is no speedup with this solution.

2.3.3 Offset trick

The dominant term in the precomputation time is $(n+1)^{2t}$, as the size of σ is assumed fixed. Now consider the values in the distance table, each $D[i,j]$, with $i,j > 0$, is computed with the values from $D[i-1,j-1]$, $D[i-1,j]$ and $D[i,j-1]$. If $A[i] \neq B[j]$ then $D[i,j]$ is equal to the minimum of the three entries plus one, if $A[i] = B[j]$ then $D[i,j]$ is equal to $D[i-1,j-1]$, hence $D[i,j]$ is less than or equal to these entries plus one. Conversely, for adjacent row entries, the optimal edit distance of $A[1\ldots i]$ and $B[1\ldots j]$ is located in $D[i,j]$, by omitting $B[j]$ from the alignment the optimal edit distance for $A[1\ldots i]$ and $B[1\ldots j-1]$ is located in $D[i,j-1]$. Now if the alignment matches $B[j]$ with some character in $A[1\ldots i]$ then by omitting $B[j]$ from the alignment, the distance is increased by at most one. If $B[j]$ is not matched then its omission will decrease the distance by at most one. Hence $D[i,j-1] \leq D[i,j]+1$. The same goes for adjacent column entries. For adjacent diagonal entries, if $A[i]$ is aligned with $B[j]$ then it is clear that $D[i-1,j-1] \leq D[i,j]+1$. If $A[i]$ is not aligned with $B[j]$ then either $A[i]$ or $B[j]$ is aligned with a gap, and $D[i-1,j-1] \leq D[i,j]$. Hence two adjacent entries can differ by at most one.

With this knowledge it is easy to see that a row or column can be represented as a start value and the difference (offset) of each subsequent entry in the row or column. An offset vector is then a t-length vector of values $\{-1,0,1\}$, where the first entry must be zero. The key to make the Four-Russians speedup efficient is to compute the edit distance using only the offset vectors. Because the number of offsets is much less then the number of possible distances, making the precomputation time only $O\left(3^{2t}\sigma^{2t}t^2\right)$.

Computing the offset vector of the last row and column of a t-block can be done without any actual edit distance. Consider a t-block in the distance table where the upper left corner is $D[i,j] = S$, where S is an unknown edit distance. Then for a column, k, in the block, the value in $D[i,k]$ is then, S plus the total of the offsets in row i from column $j+1$ to k. So even though the value of S is unknown, the value of the entry can be expressed as S plus a value which is computed from the row offset vector in row i. Each $D[k,j]$ can be expressed the same way. Now let $D[i,j+1] = S+I$ and $D[i+1,j] = S+J$ where I and J is known (the offset vectors of row i and column j). If $A[i] = B[j]$ then $D[i+1,j+1] = D[i,j]$, if $A[i] \neq B[j]$ then $D[i+1,j+1]$ is the minimum of $D[i,j]+1$, $D[i,j+1]+1$ and $D[i+1,j]+1$. The comparison can be done by knowing the value of I and J, hence $D[i+1,j+1]$ can be expressed as S, $S+1$, $S+I+1$ or $S+J+1$. This way every entry in a block can be expressed as a unknown S plus a value that can be determined. Since every entry involves the same variable S, the offset vector of the last row and column for a block can be determined with an abitrary value of S.

2.3.4 Applying the offset block function

To use the offset block function, cover the $(n+1)^2$ distance table with t-blocks, with overlapping rows and columns. Initialized the first row and column of the distance table and find the offset values for them. Row-wise determine the last row and column of each block. Because the blocks overlap, the last row in a block provides the first row in the block below it (if any) and the last column in a block provides the first column in the block to its right (if any). If Q is the total of the offset values computed for entries in row n, then $D[n,n] = D[n,0] + Q = n + Q$.

2.3.5 Time and space

The offset block function computes the last row and column, so each block uses only $O(t)$. There are $\Theta(\frac{n^2}{t^2})$ blocks, so the total time used when applying the Four-Russians is $O\left(\frac{n^2}{t}\right)$. Setting $t = \log n$ gives a running time of $O\left(\frac{n^2}{\log n}\right)$. If the distance table occupies quadratic space the space usage is then $O\left(n^2 + 3^{2t}\sigma^{2t}t\right)$ and $O\left(n + 3^{2t}\sigma^{2t}t\right)$ for distance table in linear space.

3 Implementation and experimental Setup

This section will describe the implementation and experimental setup. The reason for this section is because there are some part of the implementations, which is the same for both the quadratic and linear space Four-Russians speedup. Data handling, as in representation of the data used, see section 3.1.1. Managing sequences, where the offset block function can not be applied on the whole distance table, see section 3.1.3. For preprocessing of t-blocks, see section 3.1.4.

The experiments are very similar in both the quadratic and linear space. The only major difference are the limitation on input size for the quadratic space. See section 3.2.1 for specification of the computer used in the experiments. Test data and parameters as in t-block size and input size n and the reason for these choices, see section 3.2.2. A discussion on what to expect from them, see section 3.2.3.

In the rest of the report, sequence A refers to the sequence represented across the rows with size n, and variable i corresponds to a row in the distance table. Sequence B refers to the sequence represented across the columns with size m, and variable j corresponds to a column in the distance table.

3.1 Implementation

Four different types of implementations were made for sequence alignment. A standard dynamic programming implementation in quadratic space, and one where Four-Russians speedup has been applied. See section 4 and 5 for the implementation of edit distance and edit script respectively. A standard dynamic programming implementation in linear space, and one where the Four-Russians speedup has been applied. See section 6 and 7 for the implementation af edit distance and edit space respectively. The implementations without the Four-Russians speedup will be referred to as "standard implementations".

This section will describe parts of the implementations with Four-Russians speedup in a more detailed level. First is the data handling, as in the representation of sub-sequences and offset vectors. Why the representation is appropriate, how much space it consumes and how it might affect the running time of the computation. Second is the management of sequences, when the distance table can not be partitioned into t-blocks, e.g. there are missing some rows and columns, so the offset block function can not be applied on the whole distance table. I will present a solution and discuss how this might affect the running time. Finally I describe the preprocessing, how to compute the offset block function, implementation choices and how it affect the running time.

3.1.1 Data handling

The size of the sub-sequences and offset vectors is $t-1$ which differs from what have been described in section 2.3. The first offset in every offset vector is always set to 0, so here there is no need to have a offset vector of size t. The first character in each sub-sequence can be omitted from a block since the offset vectors are given before preprocessing or computing the edit distances. The distance values in the first row and column can be computed from these offset vectors with a abitrary start value (see section 2.3.3), so the first character in each sub-sequence does not participate in the computation and therefore not needed for lookups either.

The Four-Russian speedup is about precomputing and storing information about all possible instances of a sub-problem. For all possible combinations of sub-sequences and offset vectors, in this case $(3 \cdot 4)^{2(t-1)}$ as the sub-sequences and offset vectors is of size $t-1$, there are two offset vectors associated with each instance. So $(t-1) sizeof(int)$ space is used per offset vector if implemented naively. But as t grows in size the data structure used to store the precomputed data will explode in size. As an example when $t = 5$ the size of the structure would be $(3 \cdot 4)^{2 \cdot 4} \cdot 2 \cdot 4^2$, over $12GB$ of space. The need to pack the data arises, and an idea could be to only use bits to represent the bases and offsets. For this there is the *bitset* and the $vector\langle bool\rangle$ container from Standard Template Library in C++. $vector\langle bool\rangle$ allows for dynamic resizing whereas *bitset* is fixed, boost also made a variation of the *bitset* which allows dynamic resizing.

I decided not to use any of these because with $t = 6$ the structure occupies $(3 \cdot 4)^{2 \cdot 5} B$ assuming that each entry only uses one byte which is roughly $60GB$ (section 2.3.5). I aim to test with $t = 5$ and therefore the sub-sequences and offset vectors are stored using unsigned integer instead. I could have used signed integer instead, which would not have made any difference. This way I can use bit-wise operations to pack and unpack the data. Also, when using the offset block function, the offset vector representations it returns can be stored for future lookups without having to unpack the data, see below for more details. Two bits is used to represent a base A, C, G, T and two bits to represent an offset $-1, 0, 1$. For $t = 5$ the structure uses $(3 \cdot 4)^{2 \cdot 4} \cdot 4 \approx 1,6GB$ assuming that each entry only uses four bytes, which is still runnable on a $4GB$ RAM machine.

character	2-bit	index
A	00	0
C	01	1
G	10	2
T	11	3
-1	00	0
0	01	1
1	10	2

Table 1: Table of character and offset convertion.

Each base and offset is hardcoded, as the implementation is only intended for DNA sequence with edit distance. These are kept in a *char* and *int* array, *sigma* and *offset*, used for constructing an actual sub-sequence and offset vector from a representation. Furthermore there is a table converting each of the bases into their index value, needed for constructing a unsigned integer interpretation given a sub-sequence. For offsets, the index value is the offset value + 1, in this way you can get the offset vector values directly from an unsigned integer representation by only using bit shift operations. Also you can construct an offset vector representation from the offset vector values, by using the offset value plus one.

$$..00011011 = ACGT$$
$$..10000110 = \{1, -1, 0, 1\}$$

Figure 3: Example of sub-sequence and offset vector representated as an unsigned integer.

There are only three offsets so there is a combination of bits which will not be used. For that reason a table for converting an offset vector representation to a index value is needed *offbits2int* and from a index value to offset vector representation *int2offbits*. Because when allocating the structure to store the preprocessed data only $(3 \cdot 4)^{2(t-1)}$ entries are needed. But indexing the entries to the unsigned integer representation of an offset vector will increase the size, as if there were four offsets instead of three, giving a size of $(4^2)^{2(t-1)}B$, assuming that each entry only uses one byte. But that would not be the case when packing the data using only the size needed to represent an offset vector. As $t = 5$ the bits needed to represent a offset vector is 8 and there is two of them so there is $(4^2)^{2 \cdot 4} \cdot 2$ which is $8GB$, and having a structure of this size even with a machine which is capable of testing the implementation, a large part of the structure will not be cached which leads to a poor performance because of RAM latency.

3.1.2 Reading data from a representation

Reading a character from a sub-sequence representation can be done using an index value of the sub-sequence. The representation of the sub-sequence is stored at the $2(t-1)$ least significant bits (assuming right side of the unsigned integer). So for a number, *subbits*, between $\{0, 1, \dots, 4^{t-1}-1\}$, bit shift *subbits* to the left by $wordsize - 2(t-1)$. The sub-sequence representation is now at the $2(t-1)$ most significant bits. Depending on which index i that needs to be read, starting at index 0, bit shift *subbits* further to the left by $2i$, then bit shift *subbits* to the right by $wordsize - 2$ so there is only two bits left representing the sub-sequence. Then by using the unsigned integer value of *subbits* in the array *sigma* which contains the character, the character of index i from *subbits* can be retrieved.

..	00	01	10	11	=	ACGT
00	01	10	11	..		read index 1
01	10	11	..	..		
..	..	..	..	01	=	C

Figure 4: An example of reading a character from a representation.

Reading an offset from an offset vector representation, works the same way as for reading a character from a sub-sequence. The only difference is the bit combination 11 will not appear in an offset vector representation as there are only three offsets, and the array *offset* is used instead to look-up the actual offset value.

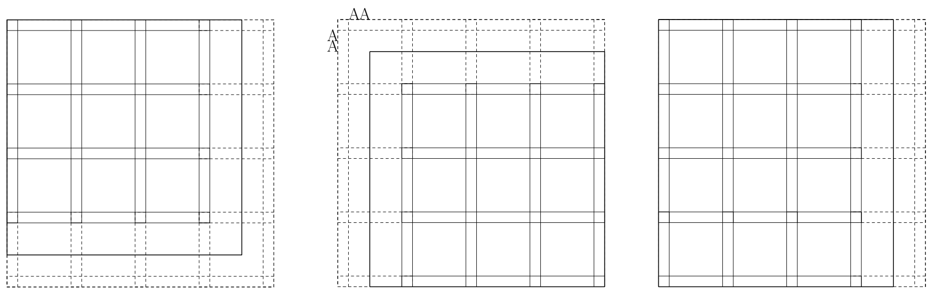
3.1.3 Managing sequences which is not a multiple of $t-1$

The idea behind Four-Russians speedup is to partition the distance table into t-blocks. A t-block has its last row shared with the first row in the t-block below it (if any) and its last column shared with the first column of the t-block on its right (if any). This means that the sequences have to be a multiple of $t-1$, if not then there are missing some rows at the bottom of the table and/or some columns to the right of the table. Consequently the offset block function can not be applied on the last rows and/or columns of the table.

One way to solve this is to use standard dynamic programming on the rows and columns where offset block function can not be applied. That means worst case having $(t-2) \times n + (t-2) \times m$ entries that uses standard dynamic programming to be filled in. This is however not very good in practise, as the time used with standard dynamic programming will be $n(t-2)+m(t-1)-(t-2)^2$ in worse case. Though it will not change the asymptotic running time of the program.

A better solution would be to make sure you can use the offset block function on the whole table. This is done by padding the sequences without changing the optimal edit distance. Furthermore one has to make sure that the entry containing the optimal edit distance is either in a row or column which is a multiple of $t-1$, so the optimal edit distance is attainable when done applying the offset block function on the whole table. Also, one needs to make sure that the optimal edit script can still be obtained with these paddings.

By letting sequences A always be the longest of two, there are two case of paddings. If n is not a multiple of $t-1$, then both sequences are padded with As in front, so that the padded sequences A now has a size which is a multiple of $t-1$. Then if m plus the size padded in front is not multiple of $t-1$, then As are padded in the back of sequences B. This way the offset block function can be applied on the whole table. The size to pad in front of each sequences, *padFront*, is given by $(t-1)-(n \mod (t-1))$, and the size to pad in the back of sequence B, $padM_{end}$, is given by $(t-1)-((m+padFront) \mod (t-1))$.



(a) When n and m is not a multiple of $t-1$.

(b) Padded As in front.

(c) When m is not a multiple of $t-1$.

Figure 5: Padding sequences.

The paddings in front do not change the optimal edit distance of the two sequences, as the optimal edit distance will go diagonal down to where the actual sequences starts, taking edit distance value 0 with it. As for the padding at the end of sequence B, they are ignored as they are only there to make sure that offset block function can be used on the whole table. So by knowing how much have been padded in the front of both sequence and in the end of sequence B, the optimal edit distance can be read from the last row in the table in entry, $n + padFront, m - padM_{end}$. As for the optimal alignment, it can still be obtained with these paddings as follows. The alignment with paddings in front has extra As aligned for both sequences, which are ignored when computing the edit script. Backtracking starts in the entry where the optimal edit distance is located, so the paddings in the end of sequences B will not participate in the backtracking.

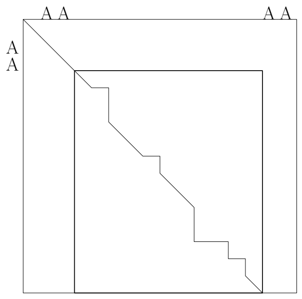


Figure 6: Edit distance and script can still be computed with padded sequences.

So what do these padding do to the running time of the algorithm? Worst case is that it is only n, the size of sequence A, which is not a multiple of $t-1$. So by padding As in front of both sequences, means that you will have to pad As in the back of sequence B, as the padded sequence B no longer is a multiple of $t-1$. This results in two extra look-ups for each row of t-blocks, which is a constant number of extra lookups. Consequently the asymptotic running time of the algorithm remains the same.

3.1.4 Preprocessing

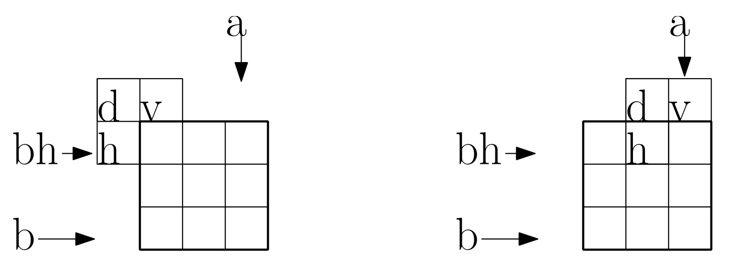
The preprocessing is about making alignments for all possible sub-sequences and offset vectors. Then save the offset vectors of the alignments in a table for fast retrieval. First, all possible sub-sequences and offset vectors are constructed. These are needed for computing the edit distance for all the sub-problems. They are not needed for other then the preproceesing step, so they are only stored temporarily. Constructing all possible sub-sequences is easy as they can generated from their index value. So by going through the numbers $0, \ldots, 4^{t-1}-1$, the sub-sequences can be generated by reading index $0, 1, \ldots, t-2$ from their index value and concatenate the characters read. For details on how to read a character in a sub-sequences representation see section 3.1.2.

The offset vectors requires a bit more work as there are only three offsets. So it is not all the combinations of two bits that is used. Constructing an offset vector is done by keeping a local unsigned integer, *offbits*, which is modified, for every number of offset vectors there is, to reflect a offset vector representation. So by going through the numbers $0, \ldots, 3^{t-1}-1$, one is added to *offbits* as long as the two least significant bits are not equal to two. If the two least significant bits are equal to two, reset them to zero by bit shifting *offbits* to the right by two. If the two new least significant bits are equal to two, reset it again by bit shifting *offbits* to the right by two. This continues until the two least significant bits are not equal to two and one is added to *offbits*. Shift the two bits, that were affected by adding one, back to their original positions. This way *offbits* now represent one of the possible offset vector. It is the same as alternating the last offset in a offset vector through all possible offset values. When reaching the end of possible offset value, 1 (2 in bit representation), the last offset value is reset and a carrier is added to the next offset in the offset vector. If this offset is the last of all possible offset values, reset it and move the carrier to the next offset value and so on. For each iteration *offbits* acts as a counter for offset vectors in their unsigned integer representations. From the constructed offset vector representation the offset vectors can be determined by reading from the representation. For details on how to read an offset value in an offset vector representation see section 3.1.2.

..	00	01	10	01	=	{-1,0,1,0} add one
..	00	01	10	10		{-1,0,1,1} add one, gives carrier, reset two bits
..	..	00	01	10		reset two bits
..	..	..	00	01		add carrier
..	..	..	00	10		move bits back to original position
..	00	10	00	00		{-1,1,0,0}

Figure 7: An example of constructing an unsigned integer representation of an offset vector.

Now all possible sub-sequences and offset vectors have been constructed and can be retrieved by their index value in a temporary storage. So finding the offset vectors of all the possible combinations of sub-sequences and offset vectors can be done by going through all the possible combinations of their index values and fetching each sub-sequence and offset vector. Then perform a standard dynamic programming computation on each combination using a $(t-1)\times(t-1)$ distance table D. By using the start value $t-1$ and then applying the offset vectors to this value, the distance table will not contain any negative distances. Not that it matters as it is only the offsets which is needed. A distance table of size $(t-1) \times (t-1)$ is only needed in the preprocessing step, as the first row and column are given by the start value and offset vectors. The only information needed for each entry is the diagonal, vertical and horizontal values from the entry. These are kept in temporary variables along with three extra variable, a to remember the first value in the last column, b to remember the first value in the last row and bh for storing the first horizontal value for a row which is used when going from a row to the row below it.



(a) Start of a standard dynamic programming computation.

(b) d,v and h moving along a row.

Figure 8: Preprocessing using standard dynamic programming.

When v reaches the last column, but is still not within the distance table, then $a = v$. The same goes for b when h reaches the last row. bh is set to h each time h starts one a new row. D is filled out the same way as in section 4.

The first offset for the last column is $D[0, t-2] - a$ and the remaining offsets are $D[i, t-2] - D[i-1, t-2]$ for $i = 1, \ldots, t-2$. The first offset for the last row is $D[t-2, 0] - b$ and the remaining offsets are $D[t-2, j] - D[t-2, j-1]$ for $j = 1, \ldots, t-2$. Constructing the offset vector representation from the offset values is done by using a temporary unsigned integer, *offbits*,

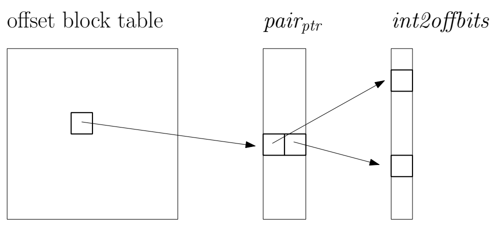


Figure 9: Mapping of offset vectors.

initialized to 0. For each offset value plus one, add it to *offbits* then bit shift *offbits* left by two and add the next offset value plus one, and so on until all the offset values have been added to *offbits*.

The offset block function is basically a four dimensional array. For each possible combination of sub-sequences and offset vectors, the offset block function returns a pair of offset vector representations which corresponds to the last row and column of a t-block. Instead of having two offset vectors stored for each entry in the offset block table, a pointer to a pair of offset vector pointers is used, see figure 9. In this way each entry of the offset block table only uses a wordsize. The pair of pointers points to the offset vector representations in *int2offbits*, which is the table used to get an offset vector representation from its index value. After constructing the offset vector representation in the preprocessing phase, the index values are also needed to set a pair of pointers saved in a array, $pair_{ptr}$. The index values are retrieved from *offbits2int* which is the mapping of a offset vector representation to its index value.

The preprocessing time is then the time to construct the sub-sequences and offset vectors, plus the time to find the alignments for all combinations of these, and plus the time it takes to read the offsets and construct the offset vector representations. The time to construct the sub-sequences and offset vectors is $O\left((4^{t-1}+3^{t-1})(t-1)\right)$. The time for aligning the combination and making the offset vector representation is $O\left(4^{2(t-1)}3^{2(t-1)}((t-1)^2+2(t-1))\right)$. But since $2(t-1) < (t-1)^2$ and $(4^{t-1}+3^{t-1})(t-1) < 4^{2(t-1)}3^{2(t-1)}(t-1)^2$ the overall time is $O\left(4^{2(t-1)}3^{2(t-1)}(t-1)^2\right)$. The space consumption is $O\left(4^{2(t-1)}3^{2(t-1)}\right)$ as each entry in the offset block table only uses four bytes, regardless the size of t.

3.1.5 Offset block function

To use the offset block function the index values of the sub-sequences and offset vectors are needed. To get these their representation needs to be constructed. After constructing the offset vector representation the index value can be retrieved from *offbits2int*. For the sub-sequences the index values corresponds to their representations.

3.2 Experiments

The objective of my experiments is to investigate whether the Four-Russians speedup for dynamic programming in sequence alignment does give a speedup in practise. So I need to construct experiments that show if this occurs with and without preprocessing. As the advantage of the Four-Russians speedup is the preprocessed data can be used on multiple pair-wise alignments. Also, for finding an edit script the Four-Russians should be able to give a speedup. In the quadratic space, the distance table has to be fill in before an edit script can be computed. With the Four-Russians speedup the distance table is only partial filled in. A backtracking on a partial filled distance table is then to go through the blocks that contain the optimal path. When the time used to compute the sub-paths in these block is less than the speedup gained by using the Four-Russians, it should yield a better running time. See section 5 for more details. For linear space, the Four-Russians speedup can be directly applied to Hirschberg's algorithm for computing optimal alignments in linear space. When the preprocessing takes less time than the time used to compute the edit script, there is a good possibility that the overall time for computing an edit script is faster when only using Hirschberg's algorithm, see section 7 for more details.

The experiments will be run with and without optimization. The reason is that the standard implementations will benefit a lot more from optimization than the ones with Four-Russians speedup. Each entry in the standard implementations is computed by comparing three other entries and two characters, so constant time is used in each entry. Each block in the implementations with Four-Russians speedup consist of computing the index values of the sub-problem, applying the offset block function and filling in the last row (and column) of the block, so the time is dependent on the size of the block. Comparing the workload of each loop it is clear that the standard implementations have less work per loop. Now consider the loop sizes. There are $n \times m$ entries to be filled in for the standard implementations and $\frac{n}{t-1} \times \frac{m}{t-1}$ blocks with the Four-Russians speedup. Although there are fewer loops with Four-Russians speedup the workload is heavier. Hence any loop optimization (loop unrolling) will benefit the standard implementations a lot more then the ones with Four-Russians speedup. As the overhead, of using the offset block function and nonlinear writes to the memory (filling in the last column of a block) for the quadratic space version, will dominate the running time with Four-Russians speedup. So it is interesting to see how the results without optimization (-O0) are compared to results with optimization (-O3).

3.2.1 Computer specification

The experiments are run on a 2.66 GHz Intel Core 2 quad (Q9450) CPU machine with 4GB RAM and 6MB cache, running Ubuntu 9.04.

3.2.2 Test data and parameters

Random sequences are used for testing in general. The reason is the contents of the sequences will not affect the standard implementations, as they do not use a lookup table like the Four-Russians speedup. So to be able compare the implementations random sequences are used. Padded sequences might have an effect on the running time for the implementations with Four-Russians speedup. As there will be more lookups with the offset block functions. So the length of the sequences will always be a multiple of $t-1$. The block size to be tested are $2, 3, 4$ and 5. Block size 1 will not make any sense since $t-1$ would be zero. For block size larger than 5 the offset block table is too big to be tested by the machines available. With the block size $2, 3, 4$ and 5, sequences which are a multiple of 60 is used as 60 is a multiple of $1, 2, 3$ and 4.

The first test is to show when we achieve a speedup using Four-Russians with and without preprocessing compared to the standard implementation. For this purpose the input size has small interval between them and the max size will be 12000. If the Four-Russians has not achieved a speedup for sequences of this size, the implementation has failed. Then a test to see the advantage of the preprocessed data used on multiple pair-wise alignments (only done for edit distance). The input sizes will be determine from a single run of sequence alignment, where the Four-Russians outperforms the standard implementation. These tests will be done for both computing the edit distance and script. A test where the sequences consist of As only to see how an alignment that have many of the same look-up would affect the running time of the implementation. Block size 5 will not be tested for the quadratic space, as the size of the distance table grows there will not be enough space for the offset block table. The test on sequences with only As will only be tested on block size 4 and 5 as the test is just to see how much faster it would be compared to random sequences. All tests are run multiple times to equalize the running time, tests on block size are done with the same two sequences.

3.2.3 Expectations

Tests with block size 2 are expected to be slower than the standard implementation. Because with block size 2 the whole distance table will be filled out and the overhead of using Four-Russians will dominate the running time. To use the offset block function for an entry the index values of the combination are needed. Whereas for the standard implementations filling in an entry is to find the maximum of other three entries, which is basically comparing three values. So constructing/loading four index values and making a lookup using the offset block function versus comparing three values, I would guess the latter case to be fastest of the two.

As for block size 3 there should be an advantage when using Four-Russians. There will be at least one entry for each block that is not filled in (intermediate results are not filled in when using Four-Russians speedup in linear space, see section 6.1 for more details). Without preprocessing the Four-Russians speedup should outperform the standard implementation later then for larger block sizes. With preprocessing the Four-Russians speedup should outperform the standard implementation earlier than for larger block sizes, as the preprocessing time is lower.

For block size 4 the preprocessing time is increased. So even though the Four-Russians speedup without preprocessing might outperform the standard implementation earlier than for block size 3, it will occur later with preprocessing. The question would then be when it is more beneficial to use block size 4 over block size 3. For large sequences there would be an advantage while for smaller sequences it might turn out to be more appropriate to use block size 3.

For block size 5 the preprocessing time will be very long and it might turn out to be useless for sequences sizes that I am capable of testing with the machines I have available. Never the less, the speedup without the preprocessing should be faster than for block size 4. The interesting part would be how much faster it is compared to the other block sizes.

The interesting part of the test with sequences consisting of only As would be to see how much faster it is compared to random sequences, as the lookup data is going to be catched. So it is expected to be the fastest of all the tests, as it is tested without any intermediate results being filled in.

For edit script tests in quadratic space, the backtracking part is slower for the Four-Russians implementation than for the standard implementation. Because, beside backtracking on the distance table, some of the entries in the table still need to fill in to be able to backtrack on the distance table. But the overall running time should be faster as the distance table needs to be filled in before this is possible. The backtracking part, even though it takes longer time than the standard implementation, will still be low and combining it with the running time to compute the edit distance will probably not result in a significant increase. For linear space, the running time should be faster than the standard implementation as it uses the offset block function while backtracking.

A quick summary, without preprocessing the larger the block size is the better the speedup is. Only block size 2 might show to be slower than the standard implementation. With preprocessing block size 3 might be best for small sequences and block size 4 for larger sequences, while block 5 might turn out to be useless. In general the tests on quadratic space are expected to be slower then the tests on linear space. Because, the distance table in linear space will always be within the cache size with the sequences sizes used, which is not the case in quadratic space.

4 Edit distance in quadratic space

The standard implementation is very straight forward. Here, forward dynamic programing is used to fill in the distance table. So for a sequence A with size n and a sequence B with size m, the distance table D is of size $n+1 \times m+1$ with $n+1$ rows and $m+1$ columns. Start by filling out the first row with $0, \dots, n$ and then the first column with $0, \dots, m$. Going through the entries row by row starting in $D[1,1]$, the entries are filled in accordingly: the entry $D[i,j]$, where $i = 1, \dots, n$ and $j = 1, \dots, m$, is equal to $D[i-1, j-1]$ if $A[i-1]$ equals $B[j-1]$. If $A[i-1]$ is not equal to $B[j-1]$ then $D[i,j]$ is equal to the minimum of $D[i-1, j-1]+1$, $D[i-1, j]+1$ and $D[i, j-1]+1$. When the whole table is filled in the edit distance can be found in $D[n,m]$. It is easy to see that the time and space are both $O(nm)$.

4.1 Four-Russians speedup

Applying the offset block function on the distance table corresponds to looping over the indexes of the table which are multiples of $t-1$. Then for each sub-problem construct the sub-sequence representations for the sub-problem. Constructing a sub-sequence representation is very straight forward. For each character in a sub-sequence look up the index value in the *char2int* table and add it to a local unsigned integer, *subbits*, which is initialized to 0. For each subsequent character in the sub-sequence, bit shift *subbits* left by two and add the index value of the character. There is no need to construct the offset vector representations, because when either the first row or the first column is a part of the sub-problem the offset vectors consist of only 1's. From how the offset vector representations are saved in *int2offbits* that offset vector representation can be found in the last index of *int2offbits* (see section 3.1.1 for details). Furthermore the offset block function returns two offset vector representations. The offset vectors of a row can be saved in their unsigned integer representations, so they can be retrieved later for lookups in the row of blocks below. By using this knowledge, there is no need to construct the offset vector representations, see figure 10.

The pair of offset vector representations that the offset block function returns are used to fill in the last row and column of the block. For details on how to read from an offset vector representation, see section 3.1.2. Subtracting one from an offset index value will give the offset value, so there is no need to look it up in *offset*, which stores them, because the index value is just the offset value plus one, see table 1. When filling in the distance table, the last offset in the column offset vector is not needed, because the last entry of the block will be filled out using the last offset of the row offset vector.

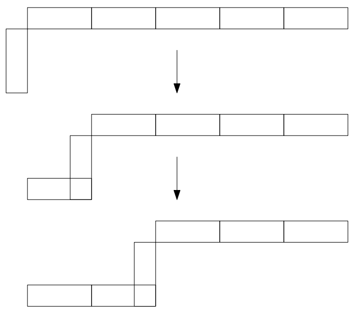


Figure 10: Remembering offset vector representations.

Padded sequences will not be a problem, as the offset block function can be applied to the whole table. Just remember that the optimal edit distance value is located in the last row minus what have been padded to the end of sequence B.

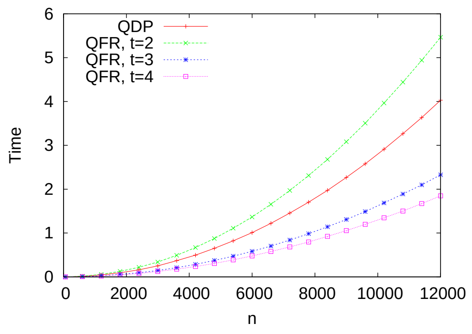
4.1.1 Time and space

Constructing the sub-sequences takes time $O(t-1)$, reading and filling in the last row and column takes $O(t-1)$ time, while getting the offset vector representations and using the offset block function is constant time as they are just array lookups. So the time used on each block is still $O(t)$ and there are $\Theta(\frac{nm}{t^2})$ blocks which gives a time complexity of $O\left(\frac{nm}{t}\right)$. The space is $O(nm)$ as there is still a table of size $(n+1) \times (m+1)$.

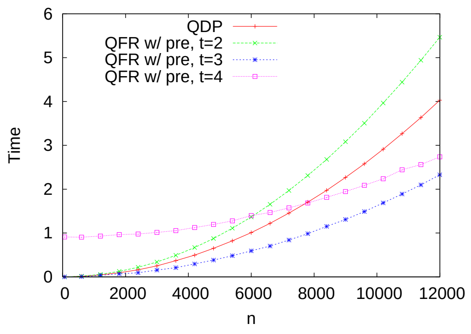
4.2 Experimental results

In this section I will show the test results regarding edit distance in quadratic space. The results without optimization is shown first then the one with optimization.

In figure 11 the experiments were run without optimization. The result of the test on block sizes $2, 3$ and 4 are shown. As expected, the running time gets faster as the block size increases where only block size 2 is slower than the standard implementation. For block size 3 with preprocessing, it outperforms the standard implementation earlier than for block size 4. For block size 3 it occurs around $n = 600$ while for block size 4 it is first at around $n = 7800$.



(a) Without preprocessing.



(b) With preprocessing.

Figure 11: Edit distance without optimization.

In figure 12 the test with optimization and witout preprocessing is shown. None of the different block sizes outperform the standard implementation. My guess is that the overhead of using the offset block function and nonlinear memory writes to fill in the columns is dominating factor. Also, the performance of all implementations is affected when the distance table is too big for the CPU cache. The standard implementation only needs to read part of the distance table into the cache to compute the edit distance, while the Four-Russians have to joggle between the distance table and the offset block table. It is clear that there are no advantages in using the Four-Russians speedup in linear space in practice.

The running time for block size 3 is not as smooth as for the others. This might be caused by the fact that for block size 3 the number of memory reads needed in each sub-problem vary depending on the input size. The variation in running time could not have been caused by a disturbance on the machine as this would also have affected the other implementations too. You do not see the same irregularity with the unoptimized implementations in figure 11(a), because the overhead of the offset block is not as dominating in these experiments.

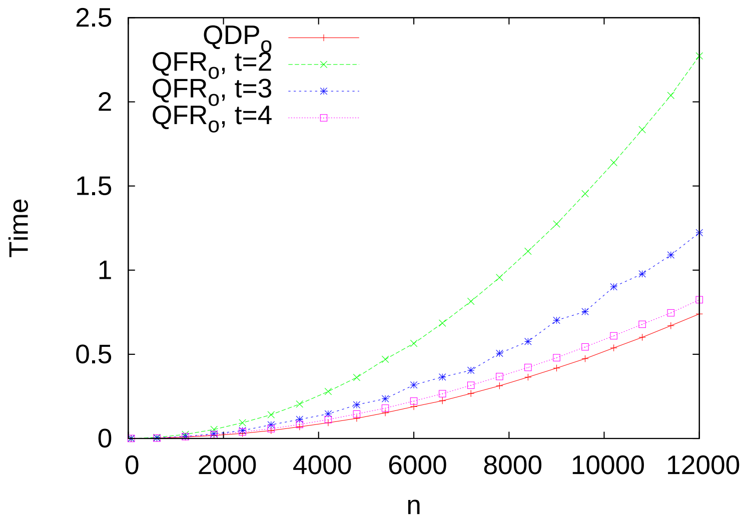


Figure 12: Edit distance with optimization.

4.2.1 Ratio

Now look at the ratio for the different block sizes with no optimization. In figure 13 it is clear that for larger block sizes the higher the ratio, meaning the better the speedup. But as you can see it is not a speedup of t. This is probably due to the overhead of using the offset block function and nonlinear writes to the memory for columns. The speedup from block size 3 to 4 is not that much, this is probably because of the offset block table. For block size 3 the offset block table is just $(3 \cdot 4)^{2 \cdot 2} \cdot 4 = 81KB$ where for block size 4 it is $(3 \cdot 4)^{2 \cdot 2} \cdot 4 \approx 11.4MB$. So for block size 4 the offset block table cannot be stored in the $6MB$ CPU cache, hence more reads from the slower main memory must be made.

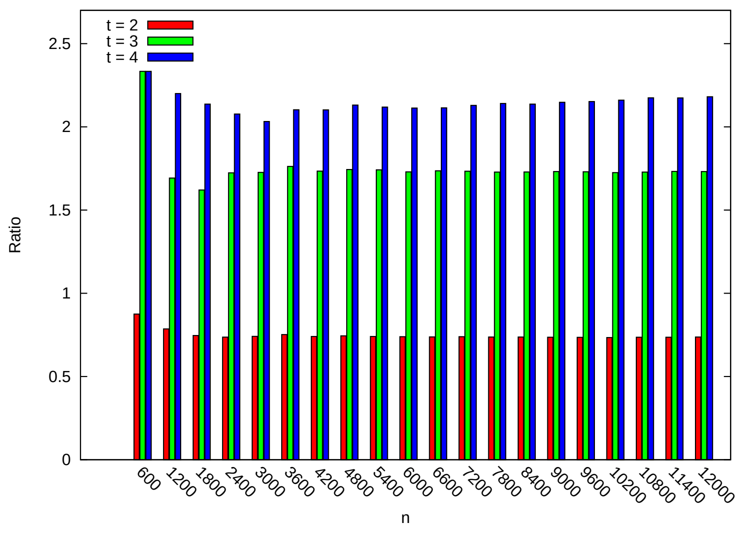


Figure 13: Ratio without preprocessing.

Taking the preprocessing time into account it is clear that for block size 2 and 3 the running time of preprocessing is close to zero. This means it will not affect the ratio. For block size 4 the ratio grows as the input size grows, which is to be expected as the preprocessing time, around 0.9 seconds, becomes less significant as the input size grows.

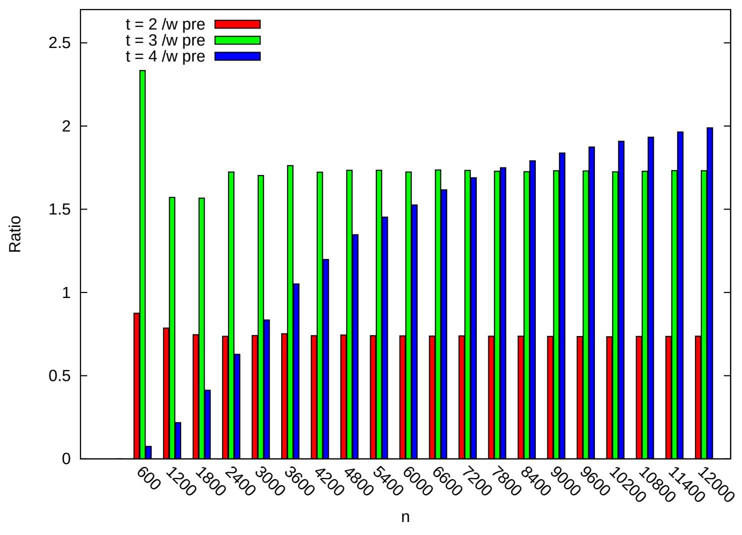
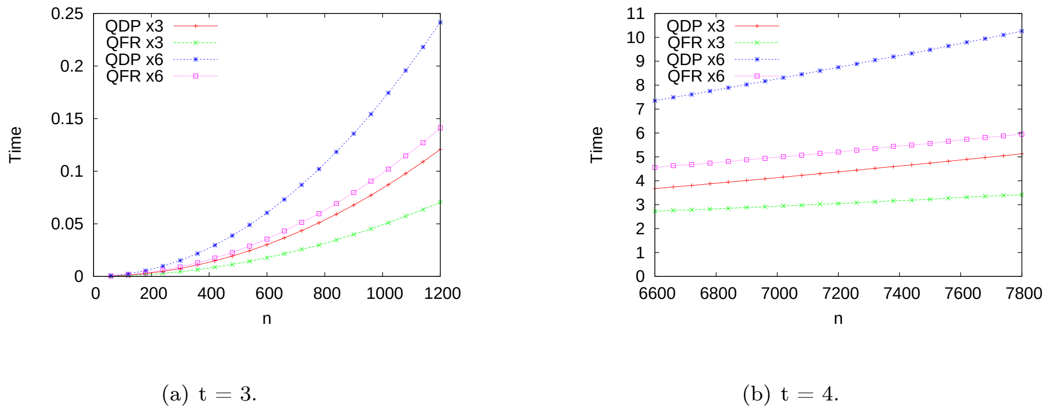


Figure 14: Ratio with preprocessing.

The ratio for the optimized test is not shown. From figure 12 none of the block sizes outperform the standard implementation so their ratio will be below one.

4.2.2 Multiple pair-wise alignments

In figure 15 you can see the tests for multiple par-wise sequence alignments. The input size interval was chosen to be where the the Four-Russians outperform the standard implementation, see figure 11(b). The running times all include preprocessing. Here one of the advantages of using Four-Russians is clear, the preprocessed data can be reused for multiple pair-wise alignments. As nice as this may look, this is still only for the non optimized implementations. Performing the same test for the optimized implementation will not show any speedup for using Four-Russians speedup with the used input sizes.



(a) t = 3. (b) t = 4.

Figure 15: Multiple pair-wise sequence alignments.

5 Edit script in quadratic space

Finding the edit script in quadratic space is done by backtracking on the distance table after it has been filled out by the edit distance function. Starting in $D[n, m]$, where the optimal edit distance is, one must find out where the value came from. For any entry i, j, if $A[i-1]$ is equal to $B[j-1]$ then $D[i, j]$ have its value from $D[i-1, j-1]$, if not then $D[i, j]$ have its value from the maximum of $D[i-1, j-1]+1$, $D[i-1, j]+1$ and $D[i, j-1]+1$.

When $D[i, j]$ is equal to either $D[i-1, j-1]$ or $D[i-1, j-1]+1$ it means that an optimal path is diagonal and $A[i-1]$ can been aligned with $B[j-1]$. If $D[i, j]$ is equal to $D[i-1, j]+1$ it means that an optimal path is vertical and $A[i-1]$ can be aligned with a gap. And if $D[i, j]$ is equal to $D[i, j-1]+1$ an optimal path is horizontal and a gap can be aligned with $B[j-1]$.

A recursive call to the backtracking method is made to find the next optimal path. When reaching $D[0, 0]$ an optimal alignment of the two sequences has been found. The running time is $O(n+m)$ as worst case would be to have only gaps aligned with both the sequences. The total time is then $O(nm)$ as the distance table needs to be filled in before backtracking.

I decided to make the backtracking recursive instead of iterative, because it is easier to time when running experiments. I am well aware of that for very large sequences the call stack will be very large and slowdown the running considerately. But for the input sizes that I going to use for the experiments, the call stack will not be large enough to affect the overall running time.

5.1 Four-Russians speedup

The distance table is only partially filled when applying the Four-Russians speedup (the last row and column of a block). So there are entries in the distance table that needs to be computed before an actual backtracking can occur. The easiest solution would be to just fill in the missing entry values for the blocks, where the optimal path goes through. Start by backtracking from the entry where the optimal edit distance is located. For $D[i, j]$ check to see if $D[i-1, j-1]$, $D[i-1, j]$ and $D[i, j-1]$ have not been computed yet. This is done by initializing each entry to the maximum value of unsigned integer. If one of these entries have not been computed, go to the top left corner of the block and start computing the edit distance using standard dynamic programming until reaching the entry where the backtrack was called from. As the block is now filled in, normal backtracking can occur within the block, until reaching a new block which is only partial filled. Done repeatedly this will compute the same optimal alignment as the standard implementation.

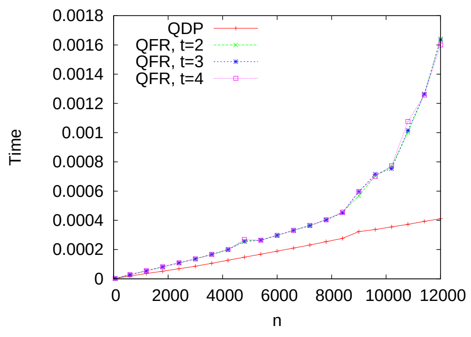
5.1.1 Handling padded sequences

The backtracking starts at the entry with the optimal edit distance, so paddings in the end of sequences B will not be a part of the backtracking. As for the paddings in front, stop backtracking when the start of both the original sequences is reached.

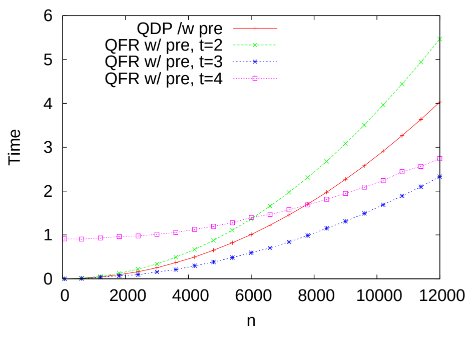
5.1.2 Time and space

Standard dynamic programming is only used on the blocks where the optimal path goes through. This gives in worst cast $\frac{n}{t-1} + \frac{m}{t-1}$ blocks where standard dynamic programming is used. For each block only $(t-2)^2$ entries need to be computed and then there is the normal backtracking $O(n+m)$ time. Total time used on standard dynamic programming is $O\left(\frac{n}{t-1} + \frac{m}{t-1}\right)$ since $(t-2)^2 < \frac{n}{t-1} + \frac{m}{t-1}$. The total time for computing the edit script is then $O(n+m)$ since $\frac{n}{t-1} + \frac{m}{t-1} < n+m$.

5.2 Experimental results



(a) Backtracking only.



(b) Overall running time.

Figure 16: Edit script without optimization.

In figure 16 you can see the result of experiments for edit script without optimization. As seen in figure 16(a), backtracking takes longer time when the distance table is only partial filled. The running time for Four-Russians is not constant like for the standard implementation and it is almost the same for all block sizes. I suspect that this is caused by the workload in the Four-Russians implementation. Besides normal backtracking the Four-Russians implementation also computes entries that have not been filled in yet. This extra work is possibly more dominant when the input sizes is increased and affecting the running time, but not enough to affect the overall running time, see figure 16(b). The running time is almost zero for only the backtracking part, so the total time to compute the edit script is almost the same as only computing the edit distance.

The ratio for edit script is sightly lower than for edit distance. As the running time is almost zero, you can not see any difference with a plotted graph so I decided not to include it. As for the test with optimization, since there is no speedup for computing the edit distance in quadratic space, there certainly will not be any speedup for the backtracking either.

6 Edit distance in linear space

Computing the edit distance in linear space is almost the same as in the quadratic space version. The only difference is that the distance table only consist of two rows. When computing the edit distance an entry in the distance table $D[i,j]$ only needs to know the diagonal value $D[i-1,j-1]$, vertical value $D[i-1,j]$ and the horizontal value $D[i,j-1]$. So when computing a row in the distance table the only values needed is in the row above the current row and a horizontal value. Which means that you only need one row and an extra variable to hold the horizontal value to compute the edit distance. One extra row is needed when computing the edit script, as you need to find out where each entry got its value from (see section 7). Filling in the distance table is done almost the same way as in section 4. As there are only two rows, you will have to switch between these to compute the edit distance. This is done by letting each row, in the quadratic space version, which is a multiple of two be the first row in the linear and any other rows be the second row. This way when computing a row the other row will contain the values needed to fill in the row. More specific $D_q[i,j] = D_l[i \mod 2, j]$, where q is the quadratic distance table and l is the linear distance table.

$$D[i \bmod 2, j] = \begin{cases} D[(i-1) \bmod 2, j-1] & \text{if A[i-1] = B[j-1]} \\ \max \begin{cases} D[(i-1) \bmod 2, j-1] + 1 \\ D[(i-1) \bmod 2, j] + 1 \\ D[i \bmod 2, j-1] + 1 \end{cases} & \text{else} \end{cases}$$

Figure 17: Filling in the distance table for edit distance in linear space.

6.1 Four-Russians speedup

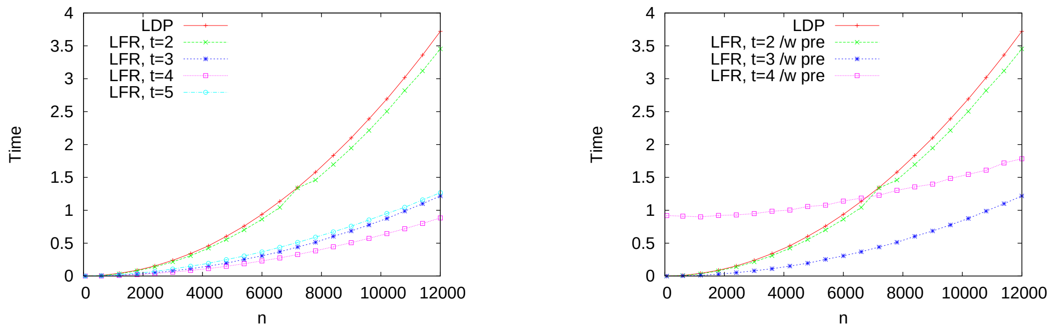
Using the Four-Russians speedup on a linear space distance table is done in almost the same way as in the quadratic version. But here you can not just take the modulus of the row index like in the standard implementation. Depending on the block size, the first row and last row in a block might give the same row index in the linear distance table. An example would be for block size 3. The first row in a block would be a multiple of 2 but so will the last row of the block. The solution is to keep a counter which is initialised to zero for the row index of the current row. By adding one and then computing the counter modulus two the index of the next row can be determined. This counter has to be stored after the Four-Russians speedup has been applied, as it is needed to locate the optimal edit distance. It is no longer necessary to fill in the last column of a block, which removes all nonlinear memory writes thus increasing performance. Also, only the last row of the computation is needed to retrieved the optimal edit distance, so there is no need to fill in any intermediate results when applying the Four-Russians speedup. This will improve the performance even more.

6.1.1 Time and space

The space consumption is $O(m)$ for the distance table and $O\left((3 \cdot 4)^{2(t-1)}\right)$ for the offset block table. As the time used on filling in intermediate results is gone, the time used on each block now consists of the time needed to identify the combination and the time used by the offset block function. The time needed to identify the combination remains $O(t)$ as for the quadratic space. There are still $\Theta(\frac{nm}{t^2})$ blocks to compute, with $O(t)$ time spent on each block, so the time is $O\left(\frac{nm}{t}\right)$.

6.2 Experimental results

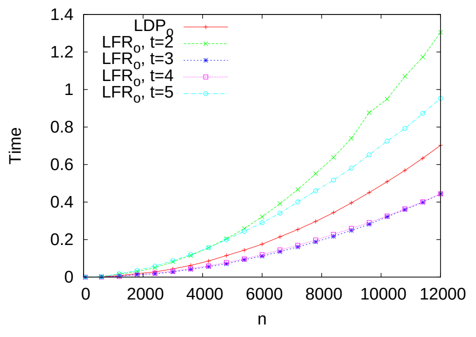
In this section I will show the test results for the edit distance in linear space. First without optimization then with optimization.



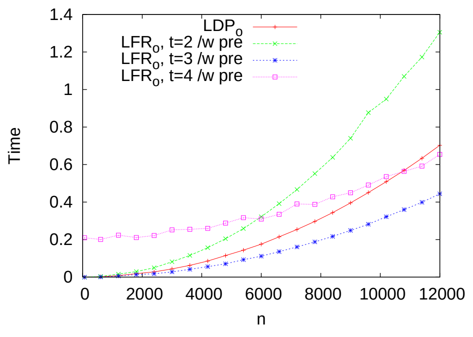
(a) Without preprocessing. (b) With preprocessing.

Figure 18: Edit distance without optimization. t = 5 with preprocessing is not plotted as the preprocessing time is approximately 210 seconds and would dominate the overall running time.

Surprisingly block size 2 is now slightly faster then the standard implementation. This is probably the result of not having to fill in any intermediate result. With block size 3, 4 and 5 Four-Russians implementation outperform the standard implementation like in the quadratic space. Surprisingly block size 5 is slower than block size 3 and 4 in figure 18(a). For block size 5 the offset table is approximately $1.6GB$, so my guess is that this is caused by lack of cache memory. For block size 3 and 4 where the offset block table is less than $11.4MB$ most of the table fit in the CPU cache. The influence of RAM latency will become more clear when I present an experiment where the sequences consist of only A, as each lookup data will be cached. This experiment will be presented in section 8.



(a) Without preprocessing.



(b) With preprocessing.

Figure 19: Edit distance with optimization. t = 5 with preprocessing is not plotted as the preprocessing time is approximately 45 seconds and would dominate the overall running time.

In figure 19 the test with optimization is shown. In figure 19(a) it is seen that only block size 3 and 4 outperform the standard implementation without preprocessing. The speedup for block size 3 and 4 is almost the same. It looks as though that with larger input sizes block size 4 will outperform block size 3. Since the workload in the Four-Russians has been reduced to identifying the combination and using the offset block function, the most time consuming factor is the overhead of using the offset block function. Here it is clear that the bottleneck of Four-Russians is the offset block table, and the bigger t gets the bigger the overhead is using Four-Russians.

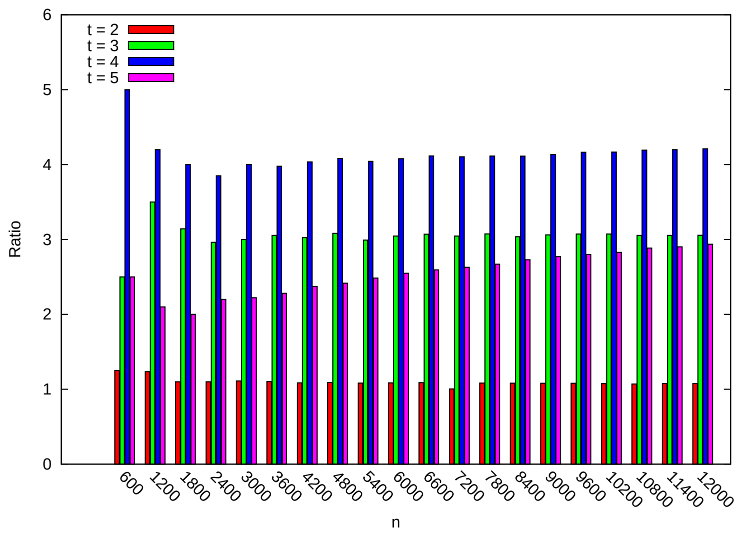


Figure 20: Ratio without preprocessing.

With preprocessing, block size 3 outperforms the standard implementation at approximately $n = 600$. For block size 4 it is first at around $n = 10800$. So with the preprocessing time taken into account, it is clear that block size 3 gives the best speedup. But if the offset block table is reduced, then block size 4 might still be more appropriate for larger sequences.

6.2.1 Ratio

In figure 20 the ratio for the two implementations without optimization and preprocessing is shown. You can see that the running time has improved as the ratio is higher than in figure 13. My guess is, since the distance table only uses linear space, the joggling between the distance table and the offset block table requires less memory reads as there are more cache space available for the offset block table. Also, the are no nonlinear memory writes anymore as the columns no longer need to be filled in. Furthermore, the intermediate results are not filled in the distance table improving the performance even more.

The ratio for block size 4 increases slightly as the input size increases. Block size 5 starts around 2 and increases to around 2.9. For block size 4 and 5, the increment in the ratio is possibly due to the overhead of the offset block function becoming less significant as the input sizes increases. You can see for block size 5 that the increment in the ratio is slightly decreasing. So for larger input size the ratio will likely converge to a constant.

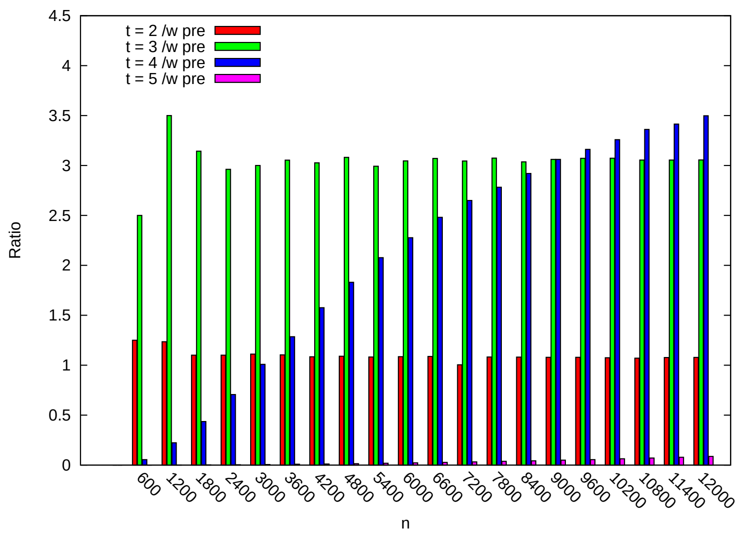


Figure 21: Ratio with preprocessing.

With preprocessing taken into account, figure 21, the ratio is a bit lower. For block size 2 and 3 there is almost no difference as the preprocessing time is almost zero. For block size 4 and 5, the ratio increases as the input size increases, because the time used for preprocessing becomes less significant.

In figure 22 and 23 you can see the ratio with optimization. As you can see with optimization and no preprocessing the ratio has dropped. For block size 3 it has gone from a ratio 3 to about the half, 1.5, and for block size 4 the ratio has gone from 4 to a little below 1.5. The overhead from using the offset block function is more dominate with optimization resulting in a poor performance for block size 4. Figure 23 shows the result with preprocessing. You can see for block size 4 the ratio converges faster towards a constant than in figure 21, this is because with optimization the preprocessing time has dropped from about 0.9 seconds to 0.2 making it less significant. This results in the ratio converging faster towards a constant. Here you can see that with and without preprocessing block size 3 gives the best speedup.

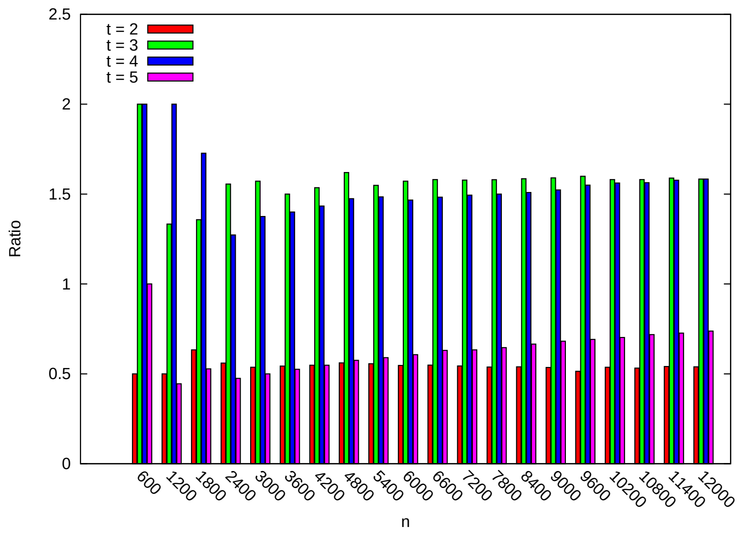


Figure 22: Ratio without preprocessing and with optimization.

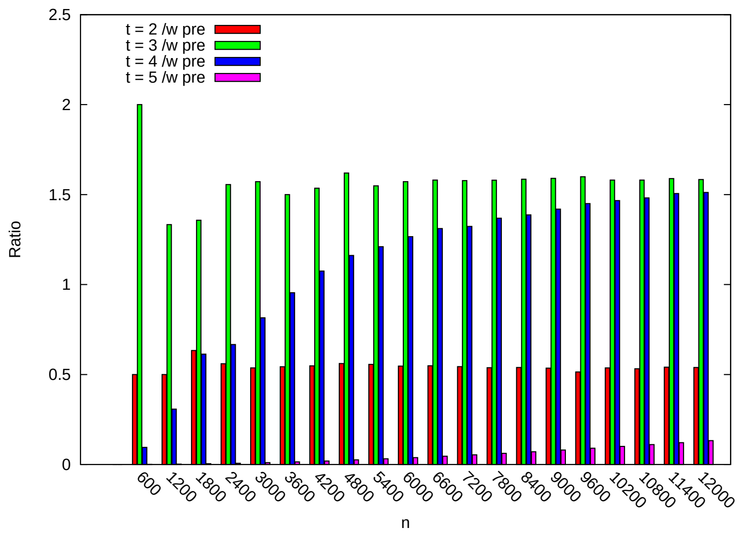
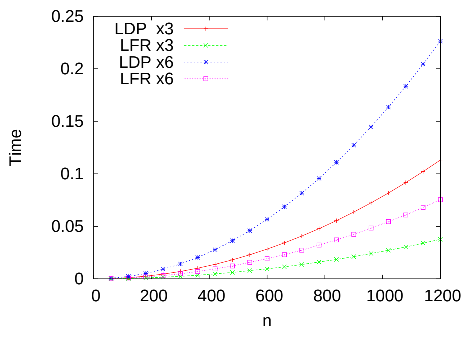


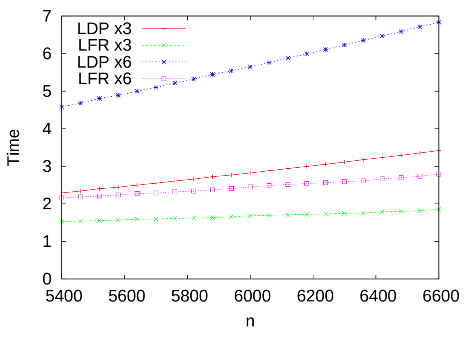
Figure 23: Ratio with preprocessing and optimization.

6.2.2 Multiple pair-wise alignments

In figure 24 you can see the test for multiple pair-wise sequence alignments without optimization. The input size interval was chosen to be where the Four-Russians outperform the standard implementation, see figure 18(b). One of the advantages of the Four-Russians speedup is the pre-processed data can be reused on multiple pair-wise sequence alignments. This test emphasizes this advantage. Block size 3 is better for small sequences whereas block size 4 is better for larger sequences.



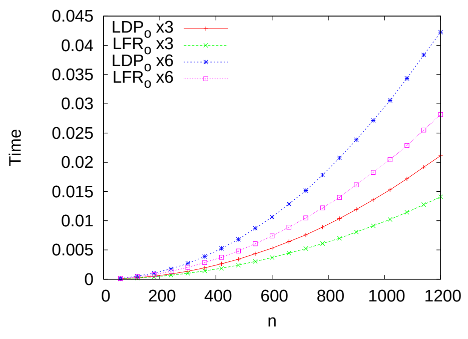
(a) t = 3.



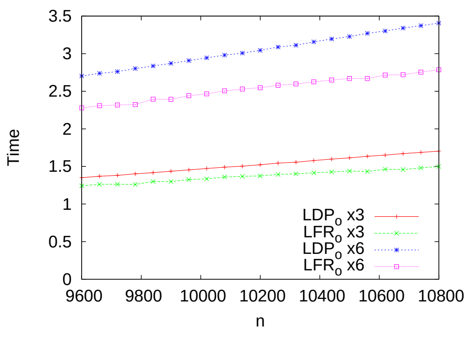
(b) t = 4.

Figure 24: Multiple pair-wise sequence alignments without optimization .

Multiple pair-wise sequence alignment with optimization can be seen in figure 25. The experiments was only done for 3× and 6× pair-wise alignment so the speedup gained is not so significant. However, if you need to align many larger sequences there is an advantage of using the Four-Russians speedup.



(a) t = 3.



(b) t = 4.

Figure 25: Multiple pair-wise sequence alignments with optimization.

7 Edit script in linear space

The distance table consist of only two rows so backtracking using only the distance table is no longer possible. For each call to the backtracking function the edit distances of the row above the current row is needed. One solution is for each row run the edit distance function down to the row where the backtracking function is called from. The time is

$$nm + (n-2)m + (n-4)m + ... + 2m =$$
$$m(n + (n-2) + (n-4) + ... + 2) = \Theta(nm^2),$$

$$\text{because } \frac{n(n-1)}{4} \leq n + (n-2) + (n-4) + ... + 2 \leq \frac{n(n-1)}{2}.$$

A more efficient solution is needed. Hirschberg [9] gave an $O(nm)$ algorithm to compute the optimal edit script in $O(m)$ space. This will be the algorithm used for the standard implementation.

7.1 Hirschberg's algorithm

In this subsection I will briefly describe Hirschberg's algorithm. The reduction of space consumption is at the cost of running time, roughly doubling the running time used in quadratic space. The main idea is based on that the optimal edit distance for two sequences A and B is the same optimal edit distance for sequences A_r and B_r, where A_r is A reversed and B_r is B reversed. Dividing the problem into sub-problems the condition still holds. More specifically, $D[n,m] = \min_{0\leq k\leq m}\{D[\frac{n}{2},k] + D_r[\frac{n}{2}, m-k]\}$, where D_r is the distance table for aligning the reversed sequences. D is referred to as the normal distance table and D_r is referred to as the reversed distance table.

Finding an optimal edit distance of sequence A and B is the same as splitting sequence A into two sub-sequences $subA_1$ and $subA_2$ and splitting sequence B into two sub-sequences $subB_1$ and $subB_2$. Then compute the edit distance ed_1 for $subA_1$ and $subB_1$ and the edit distance ed_2 for $subA_2$ and $subB_2$. The edit distance for sequence A and B is then $ed_1 + ed_2$. Since it is the optimal edit distance we need, a splitting point k is needed to minimize the value of $ed_1 + ed_2$. The splitting point can still be determine when splitting one of the sequence deterministically. So the splitting of one of the sequences can be fixed, as in always splitting sequence A and sub-sequences of A in the middle. The splitting point k is to determine where to split sequence B and sub-sequences of B.

To find k for any row mid, two rows are needed $D[mid, *]$ and $D_r[mid, *]$, where $*$ spans over the columns in the (sub-)problem. k is then the column splitting (sub-)sequence $A[i \dots 2mid]$ into $A[i \dots mid-1]$ and $A[mid \dots 2mid]$, where $2mid \leq n$ and i is the first character of a sub-sequence of A. Once k have been determined the path from the row above to the current row can be found in both D and D_r. When these sub-paths have been found, the same will be done for two smaller sub-problems, see figure 26.

The time is $O(nm)$, since

$$nm + \frac{n}{2}m + \frac{n}{4}m + ... + m =$$
$$m(n + \frac{n}{2} + \frac{n}{4} + ... + 1) = O(nm),$$

$$\text{because } n \leq n + \frac{n}{2} + \frac{n}{4} + ... + 1 \leq 2n.$$

Space usage is $O(m)$, as the distance table is in linear space. In the next subsections implementation details will be presented on how to split into sub-problems and recursively solving the sub-problems. A time and space analyse of the implementation will also be presented.

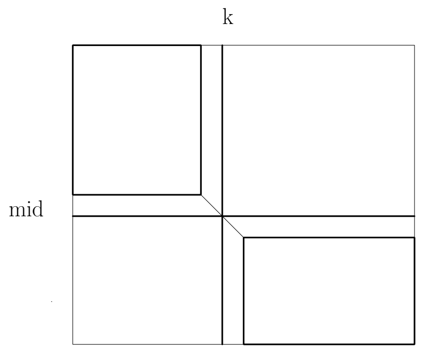
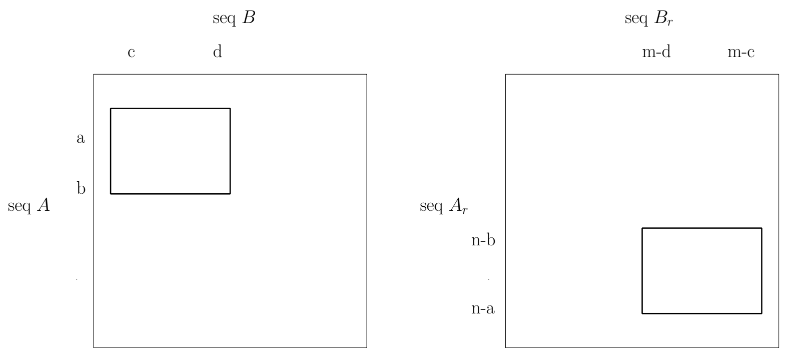


Figure 26: Hirschberg's algorithm.

7.1.1 Finding the sub-path

Instead of reversing each sub-sequence for each backtracking step, A reversed sequence of both A and B are generated before calling the backtracking function. This should lead to a slightly better running time. This solution, however, requires a bit of index fiddling as a sub-problem in the normal distance table D have to be correctly matched in the reversed distance table, to represent the reversed alignment of the two sub-sequences (see figure 27).



(a) Sub-problem in normal distance table.

(b) Sub-problem in reverse distance table.

Figure 27: Sub-problem matching.

The middle row mid for splitting (sub-)sequence A is $a + \frac{b-a}{2}$. When mid have been determined the edit distance function is called with parameters (a, mid, c, d) for D and $(n - b, n - mid, m - d, m - c)$ for D_r. When the edit distance function returns, the rows needed to determine the split point k can be found in D and D_r. As D and D_r only consist of two rows you need to find the row that corresponds to mid in D and $n - mid$ in D_r, which in this case is $mid \bmod 2$ and $(n - mid) \bmod 2$. Now, to find the splitting point k, set the minimum, $min = n + m$. Go through $j = c, \ldots, d$, if $D[mid \bmod 2, j] + D_r[(n - mid) \bmod 2, m - j]$ is less than min set the new minimum

$min = D[mid \bmod 2, j] + D_r[(n - mid) \bmod 2, m - j]$ and $k = j$. When done going through the rows the splitting point k has been determined. The optimal edit distance for the sub-problem is then located in $D[mid \bmod 2, k]$ and $D_r[mid \bmod 2, m - k]$. The optimal sub-path can then be found using $D[(mid-1) \pmod 2, *]$ and $D_r[(n-mid) \pmod 2, *]$. These sub-paths are found in the same way as in quadratic space (see section 5).

Sub-paths can be obtained from D and D_r, but as the height of the sub-problem is not always a multiple af two, there are some special case that needs to be dealt with. When $a = mid$ in D then there is no reverse sequence of the empty sequence so here D only needs to be computed to find an optimal sub-path. Here the sub-sequence of A consist of only one character, so the optimal edit distance is always in the bottom left entry of the sub-problem, then it is just backtracking from there. Another special case is when aligning the empty sequence with some sub-sequence S, here you just align each of the character in S with a gap.

7.1.2 Splitting into sub-problems.

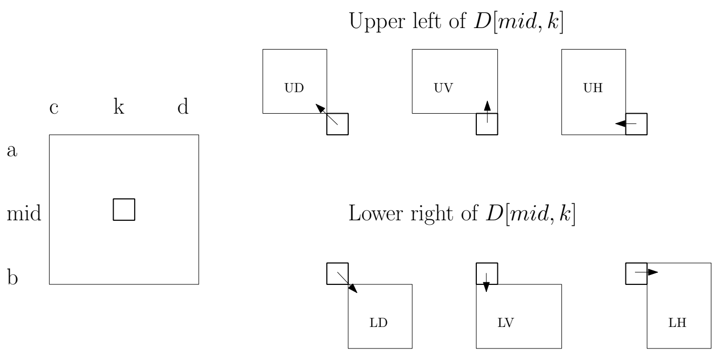


Figure 28: Split into sub-problems. U = upper, L = lower, D = diagonal, V = vertical and H = horizontal.

After each sub-path has been found recursively split the (sub-)problem into two sub-problems and run backtracking on them. The sub-problems depends on which type of sub-paths have been found. For the upper left sub-problem from $D[mid, k]$, which is the sub-path found in D. If a diagonal sub-path have been found, backtrack on sub-problem $(a, mid - 1, c, k - 1)$. In case of a vertical sub-path, backtrack on $(a, mid, c, k - 1)$ else backtrack on $(a, mid - 1, c, k)$. For the lower right sub-problem from $D[mid, k]$ which corresponds to the sub-path found in D_r. If the sub-path is horizontal, backtrack on $(mid + 1, b, k + 1, d)$. If vertical, backtrack on $(mid + 1, b, k, d)$ else backtrack on $(mid, b, k + 1, d)$.

7.1.3 Time and space

Since $nm + \frac{n}{2}m + \frac{n}{4}m + ... + m = m(n + \frac{n}{2} + \frac{n}{4} + ... + 1)$ and because $n \leq n + \frac{n}{2} + \frac{n}{4} + ... + 1 \leq 2n$, the time is $O(nm)$. Space is $O(m)$.

7.2 Four-Russians speedup

Hirschberg's algorithm uses normal dynamic programing to find the rows in D and D_r needed to determine the splitting point k and optimal sub-paths. Since Four-Russians speedup is a method for speeding up dynamic programing, it can be applied. Four-Russians speedup only computes the last row and column of a t-block, so each row, i, which is a multiple of $t-1$ is computed. But the rows needed in Hirschberg's algorithm is not always a multiple of $t-1$ and even if one of them is the other one is not. The solution to this is use offset block function down to the last row where it can be applied, and then use standard dynamic programming to fill in the rest. The idea is based on [11].

7.2.1 Backtracking using offset block function

For any row i, check to see if $i-1$ is a multiple of $t-1$. If so, use the offset block function to compute the edit distances down to row $i-1$ and compute row i using standard dynamic programming. If $i-1$ is not a multiple of $t-1$, find the last row i' where the offset block function can be applied, $i' = \lfloor \frac{i-1}{t-1} \rfloor (t-1)$. Then from row $i'+1$ down to i use standard dynamic programming. But as $t-1$ is not always a multiple of 2, you can not just use $i' \bmod 2$ to get the row index that corresponds to i' in D. However you do know how many times the offset block function will be applied on rows of blocks. Knowing that the computation will always start in row 0 in D or D_r it is $\lfloor \frac{i-1}{t-1} \rfloor \bmod 2$ (see section 6.1). This is implemented in a separate function as I wanted the top layer of backtracking to be as similar to the implementation of Hirschberg's algorithm as possible.

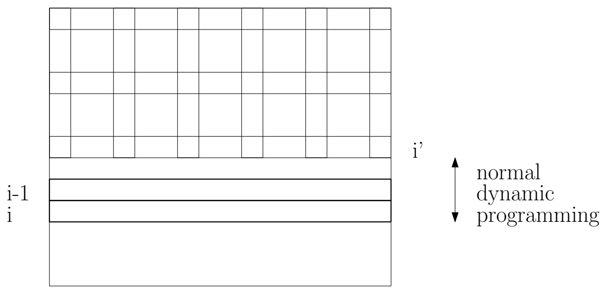


Figure 29: Combining Hirschberg with Four-Russians.

7.2.2 Handling padded sequences

Since the sequences A and B might be padded, you can not just reverse the sequences and use it when computing D_r. The actual optimal edit distance is found in the last row minus what have been padded in the back of sequence B. So when reversing the sequences the padding in front of both sequences are reversed too, since the optimal edit distance is computed with these. But the padding in the back of sequence B, are kept in the back of the reversed sequence B_r. They are only there so that the offset block function can be applied on the whole distance table, and they do not participate in the computation of the actual optimal edit distance. So when determining the splitting point k they should not be included.

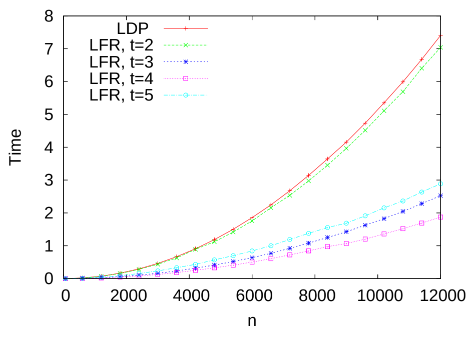
Furthermore, the length of sub-sequences of sequence B or B_r is not always a multiple $t-1$. So there might be a sub-problem starting in $\mathit{padFront} + m - 1$. Here the end padding of sequence B might need to be extended. You need to make sure that the offset block function can be applied from $\mathit{padFront} + m - 1$, and that means at least $t-2$ needs to be padded in the end. This will not affect the running time when only computing the edit distance, as the implementation will only the paddings required to find the optimal edit distance, so extra paddings in the back of sequence B will not be apart of the computation.

7.2.3 Time and space

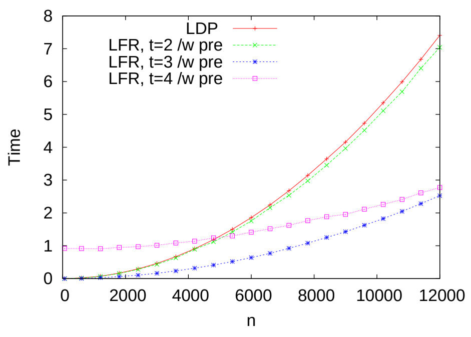
For any row i where $i-1$ is not a multiple of $t-1$ the extra time used is $(d-c)$ for row i and $((i-1) \bmod (t-1))(d-c)$ for row $i-1$. So worst case is $(t-1)m$ extra to compute i and $i-1$. Giving $O\left(\frac{nm}{t}+tm\right)$ time for any row, this gives a running time of O(nm/t) per row for $t^2 \leq n$, hence a speedup of t for any row. Since $\frac{nm+(n/2)m+(n/4)m+...+m}{t} = \frac{m}{t}(n+\frac{n}{2}+\frac{n}{4}+...+1)$ the total time is $O\left(\frac{nm}{t}\right)$. Space is $O\left(n\right)$ for the distance table and $O\left((3\cdot 4)^{2(t-1)}\right)$ for the offset block table.

7.3 Experimental results

Figure 30 shows the test results without optimization. Both block size 2 and 5 do not outperform the standard implementation. The speedup without preprocessing for the different block sizes are almost the same as for computing the edit distance. With preprocessing block size 4 outperforms the standard implementation earlier then when computing the edit distance. The reason for this is that Hirschberg's algorithm roughly doubles the running time from quadratic to linear space. Therefore, it takes longer time to compute the edit script which makes the preprocessing time less significant.



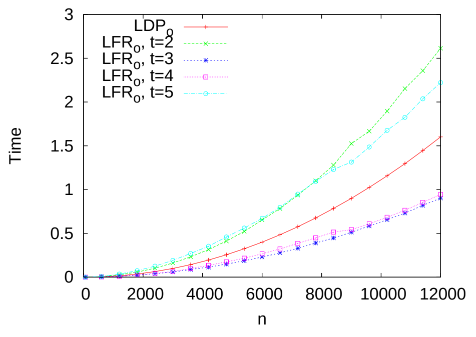
(a) Without preprocessing.



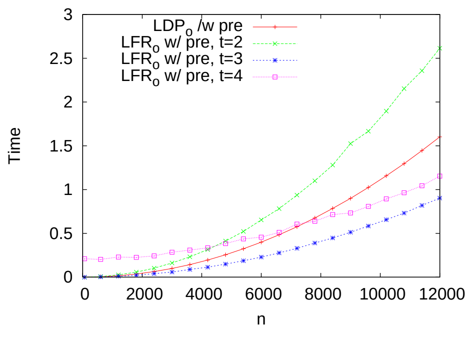
(b) With preprocessing.

Figure 30: Edit script without optimization.

In figure 31 the test results with optimization are shown. Here both block size 2 and 5 are slower than the standard implementation. For block size 2 and 5 the computation of the edit distance did not show any speedup, so combining it with Hirschberg's algorithm will also show no speedup, obviously. For block size 3 and 4 the speedup is almost the same. As in section 6.2, the only time consuming factor is the overhead from the offset block function. The overhead for block size 4 is bigger thereby making it slower than block size 3. With preprocessing, block size 4 outperforms the standard implementation earlier than for computing the edit distance, for the same reason as without optimization.



(a) Without preprocessing.



(b) With preprocessing.

Figure 31: Edit script with optimizing.

7.3.1 Ratio

Looking at the ratio for edit script computation without optimization and preprocessing, figure 32, you can see it is almost the same ratio as for the edit distance but slightly lower. This is because standard dynamic programming is used to compute the edit script, whereas for only computing the edit distance, you only use the Four-Russians speedup. With preprocessing in figure 33, the ratio is slightly better as the preprocessing time is less significant when computing the edit script.

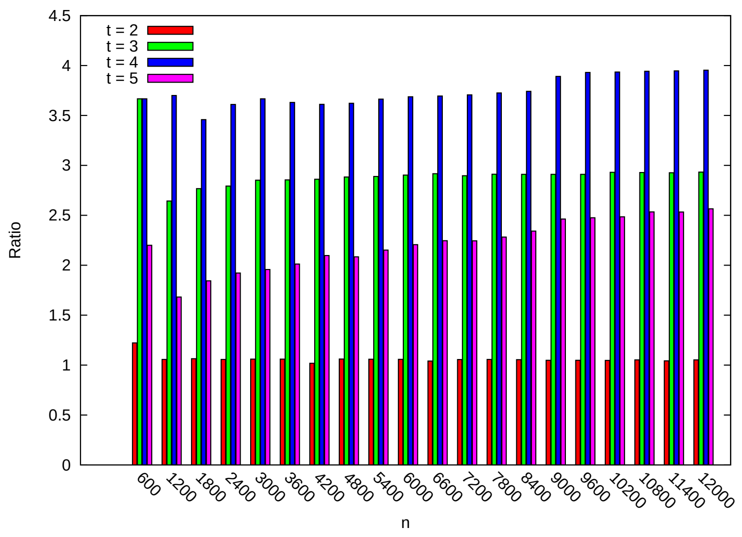


Figure 32: Ratio without preprocessing.

In figure 34 the ratio without preprocessing for edit script with optimization is shown. You can see it is slightly better than the ratio for edit distance. This is because Hirschberg's algorithm is more complex compared to only computing the edit distance. Therefore, the standard implementation of edit script does not gain as much from optimization as the edit distance. Here block size

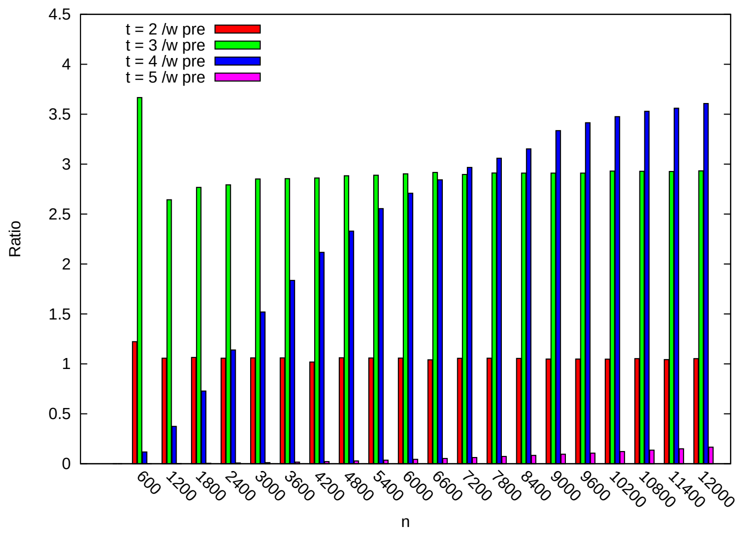


Figure 33: Ratio with preprocessing.

3 is the better choice for computing an optimal edit script. The ratio with preprocessing can be seen in figure 35. With preprocessing and optimization it is still block size 3 that gives the best speedup.

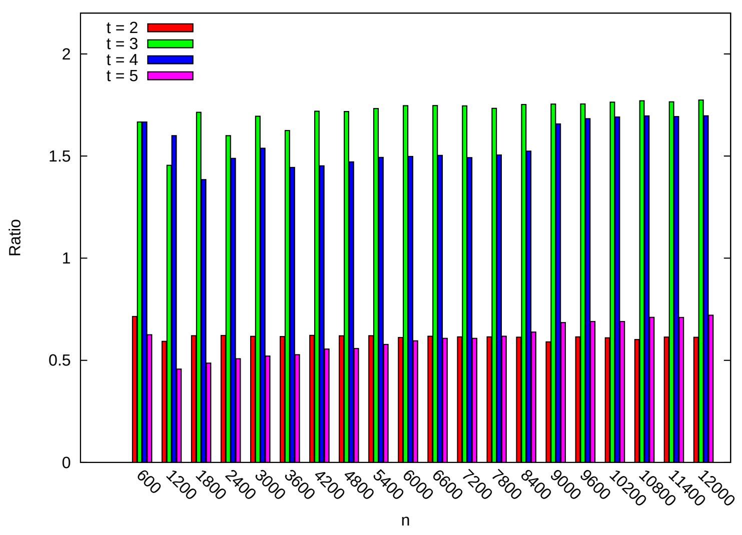


Figure 34: Ratio without preprocessing and with optimization.

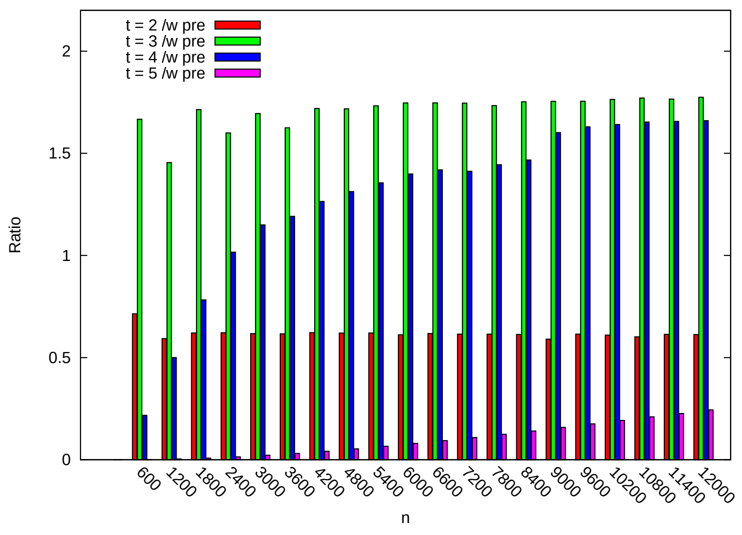
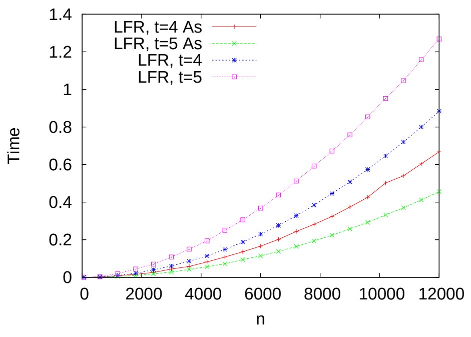


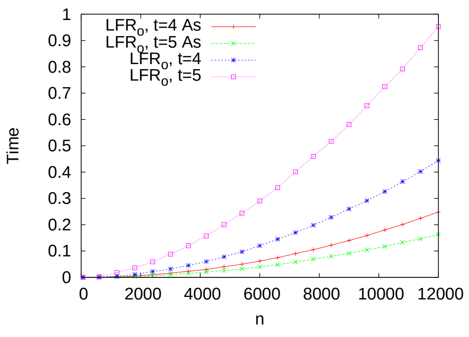
Figure 35: Ratio with preprocessing and optimization.

8 Summary and discussion

In this section I will give a summary of the experimental results and discuss the results by comparing them to each other. But before that I will show the test result for the sequences consisting of only As. In figure 36 you can see that block size 5 is actually faster then block size 4 with and without optimization. So the overhead associated with the use of the offset block function for block size 5 is the cause of the poor performance. But even without the overhead, there is still a preprocessing time which is about 210 seconds without optimization and 44 seconds with optimization. So block size 5 has proved to be useless for the input sizes used for testing. A solution might be to save the preprocessed data on a hard disk and load it when needed.



(a) Without optimization.



(b) With optimization.

Figure 36: Edit distance computations with only As and random sequences.

8.1 Quadratic vs. linear space

The input sizes I used for the experiments were max 12000 characters long. This means that when going from $O\left(n^2\right)$ space consumption to $O\left(n\right)$, the distance table became small enough to fit in the CPU cache. Consequently, freeing up more cache space for other data structures, thereby improving the performance in general.

8.1.1 Edit distance

The computational time for both standard implementations, from quadratic and to linear space, were not improved that much. The reason is that the standard implementations are very cache efficient so reads to memory is minimal. For the implementations with Four-Russians speedup, from quadratic to linear space, there were a significant reduction in the workload for each block. From having to store intermediate results to only storing the last computed row combined with more cache space for the offset block table, gave a significant speedup in the performance. That meant that instead of showing no speedup in quadratic space, Four-Russians gave a 1.6× speedup in linear space for block size 3 with optimization. This is however not a speedup of 3 which is the theoretical speedup. The offset block table is only $81KB$ for block size 3. The space consumption of the distance table and sequences is max $12000 \cdot 2 \cdot 4 + 12000 \cdot 2 \approx 117KB$. So the space consumption is within the cache size and it is no longer a RAM latency issue. But the standard implementations still reads linearly from the cache, while memory reads performed by the Four-Russians speedup are more random. So even when it is no longer a cache issue the disadvantage of the Four-Russians is still the overhead of using the offset block function. However the the Four-Russians speedup has the advantage of being able to reuse preprocessed data in multiple pair-wise sequence alignments. So for aligning multiple large sequences, Four-Russians results in a speedup in linear space.

Consider the partitioning of the distance table into t-blocks with overlapping rows and columns. This partitioning of the table can also be viewed as partitioning the distance table into $(t-1)$-blocks with no overlapping rows and columns, plus the first row and column. It is clear that the speedup is more of a $t-1$ than t. For larger block sizes the minus one becomes less significant, but for the block sizes tested it makes a significant difference. Therefore, in practice the speedup of block size 3 is closer 2× than 3×. With that in mind and considering the general overhead of using Four-Russians, the practical speedups achieved in this thesis are not far from the expected theoretical speedup.

8.1.2 Edit script

The only time increment from the quadratic to linear space is the computation of the edit script. This is because Hirschberg's algorithm uses linear space to compute the edit script, so some of the edit distances are calculated multiple times, thereby roughly doubling the computation time from quadratic to linear space. This also means that the preprocessing time for the Four-Russians speedup becomes less significant for block sizes ≥ 4. But for block sizes ≤ 3 the preprocessing times were close to zero, and yet the speedup for block size 3 is better than for computing the edit distance with optimization. This is probably because the Hirschberg's algorithm is more complex and does not gain as much from optimization as the computation of the edit distance.

9 Future work

In this section I will discuss the applicability of the Four-Russians speedup on local alignment, alignments using a general cost function and parallel programming.

9.1 Local alignment and general cost function

The objective of edit distance is to explain differences between two sequences, hence minimizing the number of gaps and mismatches. The objective in local alignment is to identify regions regions with maximum similarity, hence maximizing the number of matches. So to compute an optimal local alignment the optimization problem has to be formulated as a maximization problem rather than a minimization problem. To maximize the number of matches, a positive cost is given for matches and negative cost for mismatches and gaps. Furthermore, when computing an entry, zero has to be a part of the maximization so whenever the maximum cost of the diagonal, vertical and horizontal entry is negative the entry is set to zero. Computing a local alignment then consist of keeping a track of the entry which yields the maximum cost. This entry corresponds the end of a optimal local alignment and backtracking is needed to find the start of the local alignment.

The key to make the preprocessing fast in Four-Russians speedup was that adjacent entries could differ by at most one. This no longer applies in local alignment, as any adjacent entries can differ by n. Consequently, the number of offsets is larger which leads to more time consuming preprocessing. Furthermore, the Four-Russians only computes the last row and column of a block. So to be able to find a local alignment, information on where a local alignment in each block can be located needs to be stored along with the last row and column of the block. I therefore conclude that Four-Russians cannot be applied to local alignment.

With a general cost the number of offsets is increased, consequently increasing the preprocessing time and offset block table. I have already confirmed that for block size 5 the preprocessing time plus the overhead for using the large offset table resulted in poor performance. So here the Four-Russians speedup is not applicable either. However in [2] the authors present an algorithm with the same time complexity as the Four-Russians speedup, $O\left(\frac{n^2}{\log n}\right)$. The algorithm is general enough to support cost functions with real weights and can be applied on both global and local alignment. The algorithm described in the article is rather complex, but the idea of preprocessing the sequences is still used. It could be an interesting algorithm to investigate.

9.2 Parallel programming

Parallelization of dynamic programming in sequence alignment must handle the data dependencies presented by sequence alignment. Each entry is dependent on entries which are to the upper left diagonal of the entry. This means that entries which lies in a diagonal path from the lower left to the upper right can be computed in parallel. The same applies when using Four-Russians speedup. So blocks that lie in a diagonal path from the lower left to upper right can be computed in parallel. Furthermore the preprocessing part of the Four-Russians speedup can also be computed in parallel as the sub-problems are independent. To speedup computation even more, preprocessed each sequences in parallel so they can be used directly in the offset block function, i.e store them as their index value in the offset block function. Then the work for each block only consist of table lookups in linear space. This extension of the Four-Russians speedup seems promising and should definitely have been explored if I had had the time.

10 Conclusion

In quadratic space without optimization the Four-Russians did show a speedup in running time when computing both the edit distance and script. With optimization there were no speedup in time for any of the block sizes tested, as a result of the standard implementation being easier for the compiler to optimise and also more cache efficient. Not only is the data that it needs for computing an entry predictable, the read/write from/to memory or cache is also linear. When using the Four-Russians speedup the table usage is not predictable, and the memory access is usually nonlinear. I conclude that the Four-Russians speedup is not applicable in quadratic space.

The Four-Russians implementation did show a speedup in linear space both with and without optimization. So in practise, if you only need to align few short sequences the standard implementation is the best choice. However, if you need to aligning many larger sequences there is an advantage of using the Four-Russians speedup. Even though it is more complex to implement it might be worth the time. In conclusion, the Four-Russians speedup is applicable in linear space and yields a practical speedup.

From my experiments it is clear that the bottleneck of using the Four-Russians speedup is the size of the precomputed data, the offset block table. If the Four-Russians speedup could be made more cache efficient the performance could be significantly improved. For example for the block sizes tested, instead of packing the preprocessed data to an unsigned integer and have a pointer point to them in the offset block table, pack each offset vector as a *char* and directly save them in the offset block table. This would have reduced the table size by a factor two. So for block size 4 where the offset block table is $11.6MB$, this could mean a significant speedup as the table would now be able to fit in the CPU cache.

References

[1] V. L. Arlazarov, E. A. Dinic, A. Kronrod, M, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics Doklady*, 11(5):1209–1210, 1970.

[2] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *SODA*, pages 679–688, 2002.

[3] Elizabeth Edmiston and Robert A. Wagner. Parallelization of the dynamic programming algorithm for comparison of sequences. In *ICPP*, pages 78–80, 1987.

[4] Yelena Frid and Dan Gusfield. A simple, practical and complete $o(\frac{n^3}{\log n})$-time algorithm for rna folding using the four-russians speedup. In Steven Salzberg and Tandy Warnow, editors, *WABI*, volume 5724 of *Lecture Notes in Computer Science*, pages 97–107. Springer, 2009.

[5] Yelena Frid and Dan Gusfield. A worst-case and practical speedup for the rna co-folding problem using the four-russians idea. In Vincent Moulton and Mona Singh, editors, *WABI*, volume 6293 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2010.

[6] O. Gotoh. Alignment of three biological sequences with an efficient traceback procedure. *J. Theor. Biol.*, 121:327–337, 1986.

[7] O. Gotoh. Pattern matching of biological sequences with limited storage. *Comp. Appl. BioSci.*, 3:17–20, 1987.

[8] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[9] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[10] Xiaoqiu Huang, Ross C. Hardison, and Webb Miller. A space-efficient algorithm for local similarities. *Computer Applications in the Biosciences*, 6(4):373–381, 1990.

[11] Vamsi Kundeti and Sanguthevar Rajasekaran. Extending the four russian algorithm to compute the edit script in linear space. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (1)*, volume 5101 of *Lecture Notes in Computer Science*, pages 893–902. Springer, 2008.

[12] E. W. Myers and W. Miller. Optimal alignment in linear space. *Comp. Appl. BioSci.*, 4(1):11–17, 1988.

[13] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarity in the amino acid sequences of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[14] Stjepan Rajko and Srinivas Aluru. Space and time optimal parallel sequence alignments. *IEEE Trans. Parallel Distrib. Syst.*, 15(12):1070–1081, 2004.

[15] S. Ranka and S. Sahni. String editing on an SIMD hypercube multicomputer. *Journal of Parallel and Distributed Computing*, 9(4):411–418, August 1990.

[16] M. S. Waterman and T. F. Smith. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981.