UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

PROJECT

Fast sequence alignment

Kristijan Biščanić
Luka Hrabar
Ela Marušić

Zagreb, January 2016.

CONTENTS

1. Introduction

This project will implement, test and analyse a faster algorithm for computing string edit distances and sequence alignment. This algorithm was published by Masek and Paterson [3] and is inspired by the Four Russians Algorithm. Implemented algorithm will be compared to and tested against Needleman-Wunsch algorithm [4], which is based on dynamic programming.

The *string edit distance* is defined as the minimal cost of transforming one character string into the other. Operations allowed in those transformations are only insertion, deletion and replacing of one character, each of these having defined some cost. *Edit Script* is defined as the actual sequence of operations used to transform one string into the other. There are many algorithms that are using string edit distances and edit scripts for further calculations, and they are used extensively in bioinformatics for sequence alignment.

Sequence alignment is a process of arranging the symbolic representations of DNA, RNA or protein sequences so that their most similar elements are juxtaposed. Such alignment is useful to identify regions of similarity and many bioinformatics tasks depend upon successful alignments.

2. Algorithm

2.1. The basic algorithm

Dynamic programming problems can, most often than not, be reduced to a matrix of solutions to subproblems where the value of a cell (i, j) depends only on the cells with coordinates $(u \leq i, v \leq j)$. This is the case with the edit distance and string alignment problems. An example is shown in table 2.1, with the final solution in cell (N, M).

	-	G	C	A	T
-	0	1	2	3	4
G	1	·	·	·	·
A	2	·	·	·	·
T	3	·	·	·	·
T	4	·	·	·	·

	-	G	C	A	T
-	0	1	2	3	4
G	1	0	1	2	3
A	2	1	2	1	2
T	3	2	3	2	1
T	4	3	4	3	2

Table 2.1: String edit matrix E

The first row and column of the matrix are computed using the operation costs for the corresponding character. All other values in the matrix are computed as:

$$E_{i,j} = min(E_{i-1,j-1} + R, E_{i-1,j} + I, E_{i,j-1} + D) \tag{2.1}$$

where R, I, D are the costs of the (matrix movement) operations described in the introduction. Due to the fact that each value can be computed in $O(1)$ time in the presented manner, both the time and space complexities of this matrix calculation equal $O(N \times M)$, where N and M denote the length of the strings.

2.2. The basic algorithm with optimized space complexity

Lets say we traverse and fill the matrix from earlier in a row-major fashion. If we take a closer look we'll notice that the values from row i depend only on the values from the previous row, $i - 1$, and on the values we've already calculated in this row. Therefore, we can forget the values from row $i - 2$ and earlier, effectively reducing the memory required for the algorithm to two arrays of length equal to the number of columns. In other terms, the space complexity is reduced to $O(min(N, M))$ if we set the determine the number of columns using the shorter of the two strings in question.

2.3. The Four Russians method

We know that a problem matrix is represented using two strings and two initial vectors of values as shown earlier. In fact, every subproblem of our problem is represented using such strings and vectors:

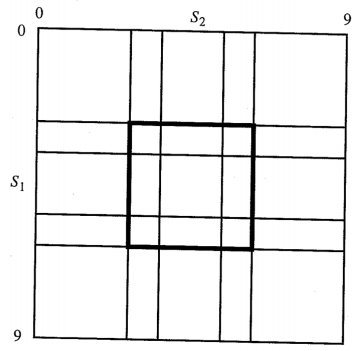

Figure 2.1: Submatrices

If we split the matrix into smaller submatrices (2.1) and precalculate the solutions of these subproblems we can combine them to solve the large matrix. This is possible because the final vectors of some submatrix are the initial vectors of some other submatrices as seen in (2.1). The final solution is yet again in the bottom-right cell of the large matrix.

The space and time complexities using this technique would be reduced to $O(min(N, M)/t)$ and $O(N * M/t^2)$, respectively. The t denotes the dimension of the submatrices used. But, in order for it to work, we need to precalculate every submatrix that might occur in our problem.

2.4. Precalculating the submatrices

The number of initial submatrix setup combinations is determined using the following expression:

$$combinations = |\sigma|^2 t \cdot |\lambda|^2 t \tag{2.2}$$

The expression is derived from the fact that every subproblem is determined using two initial vectors of costs and two strings. $|\sigma|$ denotes the size of the alphabet used for the strings, t represents the submatrix dimension, and $|\lambda|$ is the size of the set that contains every value that can occur in the initial vectors.

2.5. The step observation

The set of possible values from earlier can obviously be pretty large and we need to find a way to reduce the number of precalculations. Fortunately, we have the following observation - the values in these initial vectors, i.e. the values in general, tend to increase as we move towards the bottom-right corner of the matrix.

Before we continue, we will fix the operation costs as follows, in the context of string alignment:

- move right in the matrix, i.e. use the next character of the top string, will have cost 1
- move down in the matrix, i.e. use the next character of the left string, will have cost 1
- move diagonally down-right, i.e. use both strings' next characters will have cost 0 if the characters match, 2 otherwise

With these costs, the difference between any two adjacent matrix elements is restricted to $\{-1, 0, 1\}$ (for proof see [1]).

The next step is to code all matrix values using these differences. We have two expressions, one for vertical steps and one for horizontal steps (for more details see [3]):

$$\delta_{i,j} - \delta_{i-1,j} = min\{R_{i,j} - (\delta_{i-1,j} - \delta_{i-1,j-1}), D, I + (\delta_{i,j-1} - \delta_{i-1,j-1})\} \tag{2.3}$$

$$\delta_{i,j} - \delta_{i,j-1} = min\{R_{i,j} - (\delta_{i,j-1} - \delta_{i-1,j-1}), D + (\delta_{i-1,j} - \delta_{i-1,j-1}), I\} \tag{2.4}$$

D and I are the the costs of deletion and insertion, which are 1 in our case. $R_{i,j}$ denotes the cost of replacing the i-th character of the first string with the j-th character of the second and is equal to 0 if the two characters match (2 otherwise).

We are now able to reduce the number of submatrices we have to precalculate using vectors of steps, one horizontal and one vertical, instead of vectors of matrix values to:

$$combinations = |\sigma|^2 t \cdot 3^2 t \tag{2.5}$$

To save space we will be storing only the final vectors for each submatrix, mapped to the initial vector and string combination. The time complexity to calculate a single submatrix's final step vectors is $O(t^2)$.

2.6. Combining everything

Once we have computed all the possible submatrices, we can use them to generate the edit matrix of steps. More precisely, since we have two types of steps (those moving horizontally and those moving vertically), we will be generating two edit matrices of steps.

The first row and column of the matrix are initialized as demonstrated in the example below. The initial cost in the top left cell is 0, and the cells in the column and row are filled with the cost of deletion and insertion, respectively.

	-	G	C	A	T
-	0	1	1	1	1
G	·	·	·	·	·
A	·	·	·	·	·
T	·	·	·	·	·
T	·	·	·	·	·

	-	G	C	A	T
-	0	·	·	·	·
G	1	·	·	·	·
A	1	·	·	·	·
T	1	·	·	·	·
T	1	·	·	·	·

Table 2.2: Matrices of horizontal and vertical steps

The rest of the matrix can now be filled with submatrices we have already calculated. We know the first row and column and the substrings for the top-left submatrix, so we can simply retrieve the precalculated final row and column.

	-	G	C	A	T
-	0	1	1	1	1
G	·	·	·	·	·
A	0	−1	1	·	·
T	·	·	·	·	·
T	·	·	·	·	·

	-	G	C	A	T
-	0	·	0	·	·
G	1	·	−1	·	·
A	1	·	1	·	·
T	1	·	·	·	·
T	1	·	·	·	·

Since the submatrices in our matrix overlap, the last column of a submatrix is also the first column of the submatrix to the right (and the last row is also the first row of the submatrix below). Knowing this, we can continue to determine the last row and column of submatrices in a rowwise manner.

	-	G	C	A	T
-	0	1	1	1	1
G	·	·	·	·	·
A	0	−1	1	−1	1
T	·	·	·	·	·
T	0	−1	1	−1	−1

	-	G	C	A	T
-	0	·	0	·	0
G	1	·	−1	·	−1
A	1	·	1	·	−1
T	1	·	1	·	−1
T	1	·	1	·	1

The edit distance is the sum of steps along some path from the top left cell to the bottom right. For example, if we sum the steps along the first column and the last row, we get an edit distance of 2. The sequence of edit operations can be retrieved, provided that we have kept in memory all of the step rows and columns. We can now backtrack through the matrix by regenerating the submatrices which are crossed by the edit path. This is possible since the value in the top left cell of each submatrix can be computed in the same manner as the final edit distance, and the steps along the top row and left column are known.

2.7. Complexity analysis

If we set the submatrix dimension as $t = log_{3|\sigma|}N$, the total time complexity is equal to:

- $O(N \cdot log^2_{3|\sigma|}N)$ for submatrix precalculation
- $O(N^2/log_{3|\sigma|}N)$ for submatrix combination

The second complexity assumes the submatrices are stored in some tree-like fashion and it takes $O(log_{3|\sigma|}N)$ time to retrieve a submatrix. If submatrices were to be indexed (e.g. with a perfect hashing algorithm) and stored in an array the complexity would be reduced by a factor of $log_{3|\sigma|}N$. For more details see [1].

3. Test results

Our algorithm was tested on synthetic sequences. Pairs of sequence reads were simulated using wgsim tool from reference chromosome of Escherichia coli bacteria, and then converted into FASTA file format which is used as an input file format. Output file format of our program was MAF (*Multiple Alignment Format*) containing score, which is equal to string edit distance, and sequence alignment for algorithms that support it.

Pairs of sequence reads were generated with lengths of 100, 200, 500, 1 000, 5 000, 10 000, 50 000, 100 000, 500 000 and 1 000 000 characters. Every pair was then tested on our algorithm, as well as Fischer-Wagner algorithm for reference. Every test was timed and maximal memory usage was observed for each of the algorithms. Fischer-Wagner algorithm was tested on sequences up to 100 000 characters because of its large time complexity. Both algorithms were tested in same conditions on the same computer running inside Bio-Linux-8 virtual machine with access to 2 CPU cores and 3GB RAM.

N	Fischer-Wagner	Masek-Paterson	Masek-Paterson (alignment)
100	< 0.01 s	< 0.01 s	< 0.01 s
200	< 0.01 s	0.07 s	0.07 s
500	< 0.01 s	0.07 s	0.07 s
1000	0.03 s	0.07 s	0.09 s
5000	0.78 s	0.21 s	0.55 s
10000	3.12 s	0.65 s	1.99 s
50000	94.94 s	22.75 s	$-^{\dagger}$
100000	380.24 s	41.93 s	$-^{\dagger}$
500000	$2 - 3$ h*	724.78 s	$-^{\dagger}$
1000000	$9 - 10$ h*	49.73 min	$-^{\dagger}$

Table 3.1: Time comparison

N	Fischer-Wagner	Masek-Paterson	Masek-Paterson (alignment)
100	6976 KB	6320 KB	6832 KB
200	7008 KB	7920 KB	8448 KB
500	7024 KB	7936 KB	10.75 MB
1000	7056 KB	7952 KB	19.5 MB
5000	7296 KB	8176 KB	295.5 MB
10000	7616 KB	8416 KB	1.13 GB
50000	9824 KB	335.7 MB	–[†]
100000	12.42 MB	356.9 MB	–[†]
500000	–[*]	370.6 MB	–[†]
1000000	–[*]	392.8 MB	–[†]

Table 3.2: Maximal memory consumption comparison

[*] too much time required for execution

[†] too much memory required for execution

4. Conclusion

We've presented an implementation of fast string alignment as described in [3]. Using the Four Russians algorithm combined with the proposed edit matrix reduction system we've achieved a significant improvement over the quadratic string alignment algorithm. In terms of time complexity, our implementation is faster by a factor of $log_{3|\sigma|}(N)$ where $|\sigma|$ denotes the alphabet size. When discussing space complexity, we use less memory (by the same factor) but the required memory is still a bottleneck when dealing with long strings. Nevertheless, an improvement is visible - we can calculate the edit script for strings up to a million length if provided with a supercomputer, and that was not the case for the quadratic algorithm.

The method we explored leaves barely any room for space complexity optimization. With that in mind, the next potential improvement would be to research and implement a more memory-efficient idea, for example [2].

5. Bibliography

[1] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[2] Vamsi Kundeti i Sanguthevar Rajasekaran. Extending the four russian algorithm to compute the edit script in linear space. U *Computational Science–ICCS 2008*, stranice 893–902. Springer, 2008.

[3] William J Masek i Michael S Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System sciences*, 20(1):18–31, 1980.

[4] Saul B Needleman i Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.