Halmstad University course DA4002
Introduction to Algorithms, Data Structures, and Problem Solving

Report Template for Project 1: Sequence Alignment

Georg Michael Lexer <geolex13@student.hh.se>
Recep May <recmay13@student.hh.se>

October 18, 2013

Summary

We had to implement the Needleman-Wunsch Algorithm. In this report you will find an explanation of the algorithm a brief description of our source code and some output data generated with our program.

All the mandatory tasks are working properly (compiling without error or warning and also executing), also the first bonustask "kind of works" with some arguments but there is still a little bug in the right setting of the characters for the output.

Needleman-Wunsch Algorithm

Even for relatively short squences, there are lots of possible alignments. It will take you or a computer a long time to assess each alignment one-by-one to find the best alingment. The problem of finding the best possible alignment for two sequences is solved by the Needleman-Wunsch algorithm. The Needleman-Wunsch algorithm is an algortihm used in bioinformatics to align protein or nucleotide sequences. This algorithm was proposed by Christian Wunsch & Saul Needleman in 1970. The Needleman-Wunsch algorithm is mathematically proven to find the best alignment of two sequences using dynamic programming. Dynamic programming algorithms solve problems by breaking a large problem into smaller easy problems of a similar type. It stores the calculations on a matrix and uses them without calculating it again. And performs a global alignment on two sequences. This Algorithm is similar to the Smith-Waterman Algorithm but this one gives you a Local Alignment. And although Needleman-Wunsch Algorithm allows negative values on its table the other one doesn't accept numbers which are smaller then zero. The Needleman-Wunsch algorithm takes time proportion to n2 to find the best alignment of two sequences that are both n letters long. In contrast assessing all possible alignments one-by-one would take time proportional to (2n combination n). In this case we can say that this algorithm is much faster than accessing all alignments one-by-one just because n2 is smaller than (2n combination n). The Needleman-Wunsch algorithm works in the same way regardless of the length or complexity of sequences, and guarantees to find the best alignment. The idea is to use dynamic programming to eciently implement a recursion. We have given two input strings x and y, than we build a matrix F such that the entry F[i, j] is the score of the optimal alignment of x[1..i] and y[1..j]. At the top row and the left side, we must specify F[i,0] and F[0, j]. The value F[i,0] represents assigning a prex of x to a gap, so if we assume that d is the gap between cells, we should dene F[i,0]= id. Similarly down the left side F[0, j] = jd, corresponding to assigning a prex of y to a gap. Algorithm compares two strings and finds the difference between them. In our code first of all strings are checked if they contains only letters. Otherwise you are warned and asked to enter a new string which includes only letters. It doesn't matter if the letters you entered are lower or upper because they will be converted to the appropriate form to compare. Algoritm uses the propagation mechanism described below:

- +10 for perfect match
- -2 for matching a vowel with a different vowel
- -4 for matching a consonant with a different consonant
- -10 for matching a vowel with a consonant
- -5 for insertions
- -5 for deletions

The entire matrix is filled with scores and pointers using a simple operation that requires the scores from the diagonal, vertical, and horizontal neighboring cells. You will compute these scores: match scores, a vertical gap score, and a horizontal gap score. The match scores are the sum of the diagonal cell scores and the score for a match depens on which

kind of letters you are comparing (+10, -10, -4 or -2). The horizontal gap score is the sum of the cell to the left and the gap score (-5), and the vertical gap score is computed analogously. Once you have computed these scores, assign the maximum value to the cell and point the arrow in the direction of the maximum score. Continue this operation until the entire matrix is filled, and each cell contains the score and pointer to the best possible alignment at that point. When filling the matris has been done we should use a "trace-back process" to find the alignment. The trace back step determines the actual alignment (or alignments) that result in the maximum score and gives that alignment in reverse order. Process begins from the cell which is located at the lower right corner of the matrix to cell at the upper left corner. When we go back on the matrix we need to check these three possible moves:

- Diagonal: the letters from two sequences are aligned
- Left: a gap is introduced into the left sequence
- Up: a gap is introduced into the top sequence

When the trace back process done, we match the letters for each diagonal jump and put a gap for the horizontal jumps.

References:
- http://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm
- Links within the Wikipedia article

Source code

The whole program can be found in the file

`../code/needleman\_wunsch.c`

Inside we will see these functions:

```
void zero_matrix(Element *matrix[], int nrows, int ncols);
void checking_case_vowels(Argument input[], int length, int i);
void needleman(Argument input[], Element *matrix[], int nrows, int ncols);
int calc_match(Argument input[], int x, int y);
void calc_mdi(Element *matrix[], int match, int insertion, int deletion, int x, int y);
void get_backtrace(Argument input[], Element *matrix[], int nrows, int nclos);
void matrix_printing(Element *matrix[], Argument input[], int ncols, int nrows);
void free_matrix(Element *matrix[], int nrows);
int main(int argc, char ** argv)
```

From the main() mostly all the other functions are called from. Besides that the main() function is also the place were the matrix gehts initialized and the arguments get filled in the Argument struct.

```
int main(int argc, char ** argv)
```

These brings us to two important structs, the struct element is inside the matrix and holds the value of the cell and also the backtrace values (match, delete, insert) The second struct argument is used for the two input arguments and also takes care of the length of the input and wether they are vowel or constants.

```
typedef struct element
{
int value;
char bitmask[4];
} Element;


typedef struct argument
{
char *arg;
int length;
char *vowel;
} Argument;
```

All the "magic" happens in the needleman() function (see below):

```
void needleman(Argument input[], Element *matrix[], int nrows, int ncols)
{
    int match, insertion, deletion, calc;
    int x, y;

    for (x = 1; x < nrows; x++)
    {
        for (y = 1; y < ncols; y++)
```

```
        {
            // calculates the match value based on the vowel/constant inside the input.
            calc = calc_match(input, x, y);

            match = matrix[x-1][y-1].value + calc;
            insertion = matrix[x][y-1].value + (-5);
            deletion = matrix[x-1][y].value + (-5);

            matrix[x][y].value = fmax(fmax(match, insertion),deletion);

            // Do all the match/deletion/insertation calculating
            calc_mdi(matrix, match, insertion, deletion, x, y);
        }
        // sets insertion and deletion to zero for next iteration
        insertion = 0;
        deletion = 0;
    }
}
```

The match, delete and insert values, which are the most important parameters for the algorithm to work, are calculated in the

```
int calc_match(Argument input[], int x, int y);
```

function which returns those values:

- +10 for perfect match
- -2 for matching a vowel with a different vowel
- -4 for matching a consonant with a different consonant
- -10 for matching a vowel with a consonant

```
void calc_mdi(Element *matrix[], int match, int insertion, int deletion, int x, int y);
```

This fuctions set the values in my simplified bitmask() to 'm' , 'i', 'd' or just fills a space (' ') if there is no value assigned to it.

For the optional task we had to backtrace through the matrix one of the best ways. This happens in the get_backtrace() function, this function is not perfectly finished yet as there is still a bug in it. The backtrace worls for most of the arguments but sometimes it gets the last positions wrong. The rest of the program is still working properly.

```
void get_backtrace(Argument input[], Element *matrix[], int nrows, int nclos);
```

The functions zero_matrix(), checking_case_vowels(), matrix_printing() and free_matrix are kind of "helper functions. Comments on them you can find in the code itself.

```
void zero_matrix(Element *matrix[], int nrows, int ncols);
void checking_case_vowels(Argument input[], int length, int i);
void matrix_printing(Element *matrix[], Argument input[], int ncols, int nrows);
void free_matrix(Element *matrix[], int nrows);
```

Output

As there is also a Makefile included you can compile and run the program as followed:

```
make
```

```
./needleman_wunsch <argument1> <argument2>
```

The figures below 1 2 3 4 show the ouput which you get when running the programm with the different arguments.

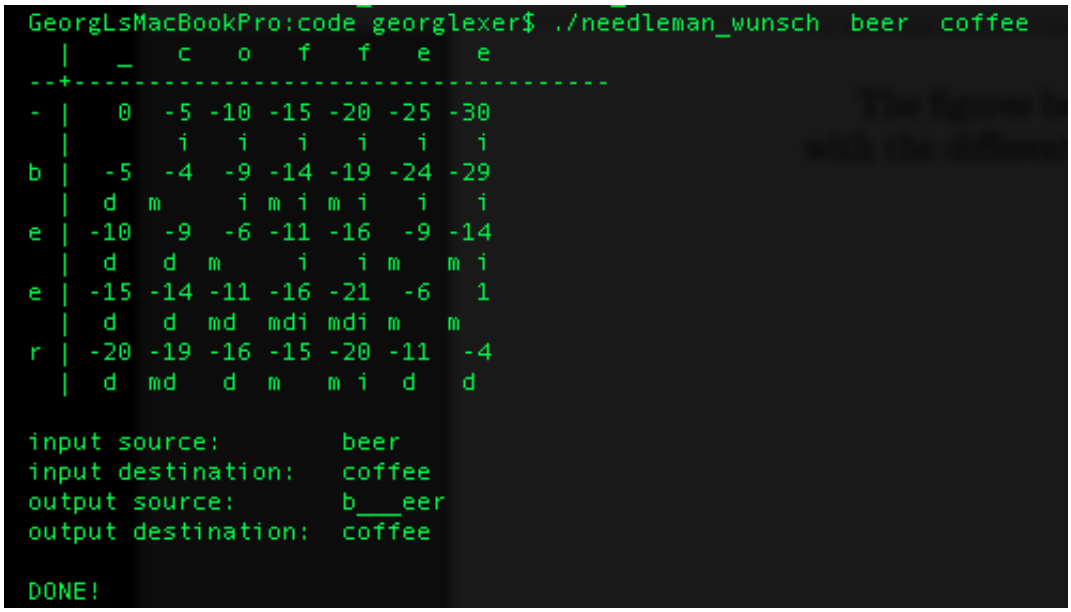

Figure 1: Output with arguments "beer" and "coffee"

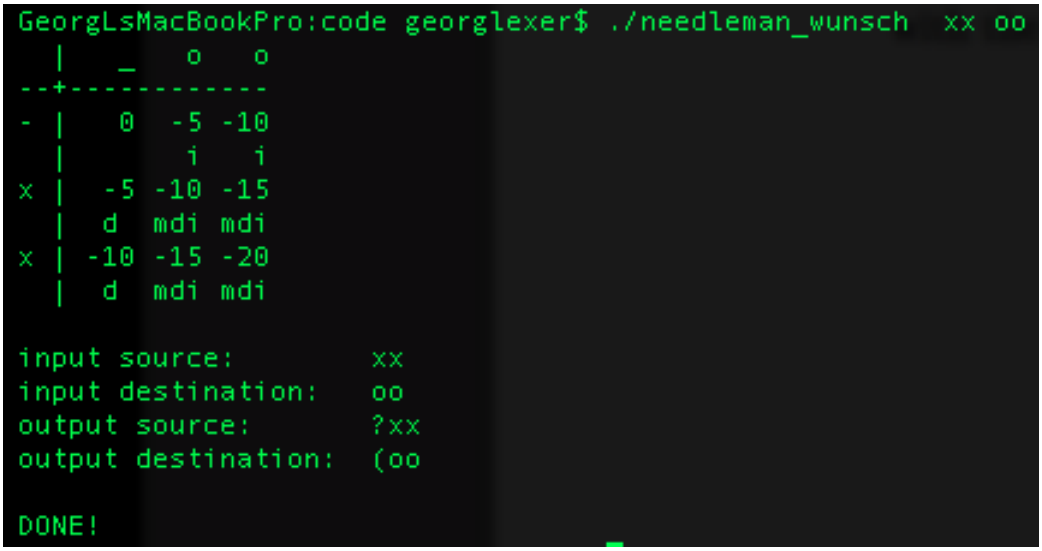

Figure 2: Output with arguments "xx" and "oo"

Figure 3: Output with arguments "hot" and "ohitishotinthere"

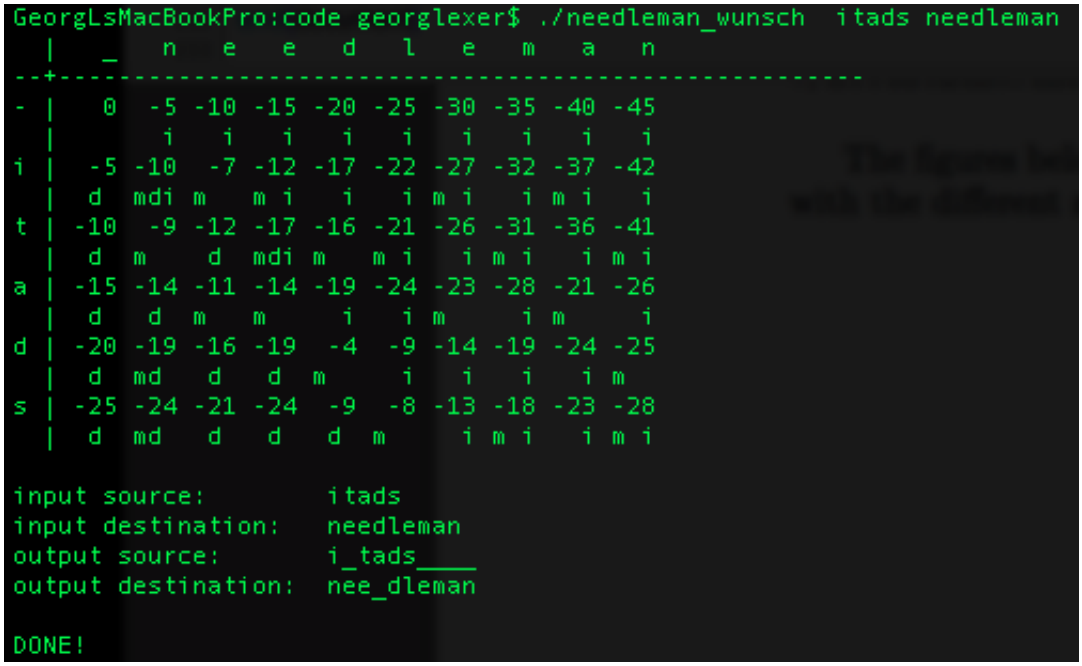

Figure 4: Output with arguments "itads" and "needleman"