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

Assignment for Project 2: Sequence Alignment

Roland Philippsen
roland.philippsen@hh.se

October 10, 2013

1 Introduction

Finding a good alignment between two sequences of symbols (characters) is a commonly encountered problem in various domains. The Needleman-Wunsch algorithm [1] presented in lecture 6 [2] efficiently finds such optimal alignments, and this project is about implementing it. The excellent article on Wikipedia [3] explain this algorithm in more detail.

In order to get started on the implementation, you are given code examples for some of the technical ingredients you will need:

- Matrices (or "tables" or "two dimensional arrays") can be implemented using either the approach in `example-matrix-a.c` or `example-matrix-b.c`. These examples also show how you might print nicely formatted output from your program.
- Your program will be required to process command line arguments, namely the two strings which should be aligned with each other. The `example-argument-parsing.c` file provides code for some of operations you will need to perform on the provided strings, such as finding the length of a string, checking that all characters are valid, and finding out whether a given letter is a vowel.

You are again required to write a report about this project, and hand it in along with your source code. Again, your project must compile and run properly on the machines in room B231c. There is no template for the report this time, but of course the report must still be well structured and contain appropriate explanations to understand what you have achieved and how.

The deadline for handing in the source code and the report is **Friday, October 18, 2013, at 18h00**. Teams who miss the deadline will receive a penalty of 5 points (the maximum number of points is 25). In case of exonerating circumstances, a deadline extension will be granted. Participants must notify the lecturer of such circumstances as soon as possible when they arise.

2 Mandatory Tasks

Create a new program that does the following things:

1. The program accepts two command-line arguments and checks that they contain only alphabetical letters.
2. It then creates a matrix of size $(N + 1) \times (M + 1)$, where N is the length of the *input source* string (the first argument) and M the length of the *input destination* (second

argument). This matrix will serve as table to store the value function (accumulated alignment scores) during the cost propagation of the Needleman-Wunsch algorithm. *Hint: you can store something other than integer values in the matrix, for example a struct. This can help to keep track of backpointer information required for a later step.*

3. After proper initialization, the value table is then filled using the propagation mechanism described on slides 28–33 of lecture 6. The scoring system should be the same as the example given in the lecture:

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

   Note that your scoring function should ignore whether the letters are upper case or lower case. Thus, for example 'R' perfectly matches 'r' with score +10, and 'A' matches 'o' with score -2. Also beware of the fact that, due to the addition of the gap character '_', table row i corresponds to letter $i-1$ of the source string, and table column j corresponds to letter $j-1$ of the destination string.

4. At the end, your program shall print the contents of the value matrix *along with backpointer information* in a properly formatted way. Some examples are given in Figure 1 which you should use to check that your program computes the correct values. *Reminder: a backpointer is information about which particular propagation action (insert, delete, or match) lead to the optimal score for a particular value in the table. There can be up to three backpointers for any matrix element.*

Make sure that your report mentions to what extent you have completed the above tasks and indicates where to find the relevant implementation in your source file (or files). The report must also include your program's output for all of the examples shown in Figure 1. As usual, it should also properly cite references (websites and other sources of information that you used).

```
input source:      beer
input destination: coffee

    _    c    o    f    f    e    e

_    0   -5  -10  -15  -20  -25  -30
          i    i    i    i    i    i

b   -5   -4   -9  -14  -19  -24  -29
     d    m    i   mi   mi    i    i

e  -10   -9   -6  -11  -16   -9  -14
     d    d    m    i    i    m   mi

e  -15  -14  -11  -16  -21   -6    1
     d    d   md  mdi  mdi    m    m

r  -20  -19  -16  -15  -20  -11   -4
     d   md    d    m   mi    d    d
```

```
input source:      xx
input destination: oo

    _    o    o

_    0   -5  -10
          i    i

x   -5  -10  -15
     d  mdi  mdi

x  -10  -15  -20
     d  mdi  mdi
```

```
input source:      kc
input destination: KC

    _    K    C

_    0   -5  -10
          i    i

k   -5   10    5
     d    m    i

c  -10    5   20
     d    d    m
```

```
input source:      HOT
input destination: ohitishotinthere

    _    o    h    i    t    i    s    h    o    t    i    n    t    h    e    r    e

_    0   -5  -10  -15  -20  -25  -30  -35  -40  -45  -50  -55  -60  -65  -70  -75  -80
          i    i    i    i    i    i    i    i    i    i    i    i    i    i    i    i

H   -5  -10    5    0   -5  -10  -15  -20  -25  -30  -35  -40  -45  -50  -55  -60  -65
     d  mdi    m    i    i    i    i   mi    i    i    i    i    i   mi    i    i    i

O  -10    5    0    3   -2   -7  -12  -17  -10  -15  -20  -25  -30  -35  -40  -45  -50
     d    m   di    m    i   mi    i    i    m    i    i    i    i    i    i    i    i

T  -15    0    1   -2   13    8    3   -2   -7    0   -5  -10  -15  -20  -25  -30  -35
     d    d    m    d    m    i    i    i    i    m    i    i   mi    i    i    i    i
```

Figure 1: Table of accumulated alignment scores, complete with backpointer information, for several examples of input source and destination strings. The letters underneath each value encode the backpointer information: 'm' stands for match *(diagonal arrow on the lecture slides), 'i' means* insert *(horizontal arrow on the slides), and 'd' means* delete *(vertical arrow).*

3 Bonus Tasks

Perform one or more of these suggested tasks to receive more points:

1. After printing the table, your program should trace back *one* optimal alignment. Remember, this starts at the bottom-right corner of the table and follows backpointers until you reach the top-left. The corresponding alignment between source and destination result in an *output source* and an *output destination* string that contain the characters from the *input source* and *input destination* but with appropriately added gap characters:

   - For a *match*, you use one character of each input source and destination, and proceed along the diagonal to the next cell in the table.
   - A *delete* puts one character from the input source into the output source, but the output destination receives a gap '_' and instead of going diagonally, the column will remain constant.
   - Conversely, an *insert* puts one character from the input destination into the output destination, whereas the output source receives a '_' and the row remains constant.

   Figure 2 shows examples of output strings that result from tracing back the example inputs shown in Figure 1.
   Implementation hint: the maximum length of the output strings is $N + M$, so you can allocate a buffer that is big enough for that plus one extra element for the terminating '\0', and fill it backwards starting at `buffer+N*M`. When printing the result, be careful to skip over the first few (unused) characters in the strings thus produced.

2. It is more challenging to trace back all optimal alignments: at every element with two backpointers, you have to create a new branch for the backtrace. Likewise, three backpointers will require two new branches. Figure 2 shows the output of a program that traces back all solutions for the the tables given in Figure 1. It is not important whether your program produces the traces in the same order, as long as it produces the same traces.

3. Create a new program that uses the Smith-Waterman algorithm [4] (a variation of Needleman-Wunsch) for sequence alignment. Compare the results it gives for the examples of Needleman-Wunsch shown in Figure 1.

References

[1] Saul B. Needleman and 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):443–53, 1970.

[2] Roland Philippsen. ITADS 2012 lecture 6: dynamic programming. `http://poftwaresatent.net/r/itads2012/slides/06-dynamic-programming-print.pdf`, October 2012.

[3] Wikipedia. Needleman-wunsch algorithm. `http://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm`, October 2012.

[4] Wikipedia. Smith-waterman algorithm. `http://en.wikipedia.org/wiki/Smith-Waterman_algorithm`, October 2012.

```
input source:      beer
input destination: coffee

output source:      ___beer
output destination: coffee_

output source:      __b_eer
output destination: coffee_

output source:      b___eer
output destination: coffee_
```

```
input source:      kc
input destination: KC

output source:      kc
output destination: KC
```

```
input source:      HOT
input destination: ohitishotinthere

output source:      ______HO___T____
output destination: ohitishotinthere

output source:      ______HOT_______
output destination: ohitishotinthere

output source:      _H_____O___T____
output destination: ohitishotinthere

output source:      _H_____OT_______
output destination: ohitishotinthere
```

```
input source:      xx
input destination: oo

output source:      xx
output destination: oo

output source:      _xx
output destination: oo_

output source:      xx_
output destination: _oo

output source:      _xx
output destination: o_o

output source:      x_x
output destination: _oo

output source:      __xx
output destination: oo__

output source:      x_x
output destination: oo_

output source:      xx_
output destination: o_o

output source:      xx__
output destination: __oo

output source:      _x_x
output destination: o_o_

output source:      x__x
output destination: _oo_

output source:      _xx_
output destination: o__o

output source:      x_x_
output destination: _o_o
```

Figure 2: Optimal alignments for the same example sources and destinations as in Figure 1. All of the listed alignments have the same score. The order in which they are listed is not relevant here, it depends on implementation details.