Introduction to Algorithms, Data Structures, and Problem Solving

DA4002 (HT12) Halmstad University

Assignment for Project 2: Sequence Alignment

Roland Philippsen
roland.philippsen@hh.se

October 11, 2012

1 Introduction

Finding a good alignment between two sequences of symbols (characters) is a commonly encountered problem in various domains. For example, it can be used to compute differneces between files, as part of spell-checking programs, and especially in bioinformatics to align protein or nucleotide sequences. A prominent example is the Needleman-Wunsch algorithm [1].

In this project, you will implement that algorithm, based on the slides of lecture 6 [2] and the excellent article on Wikipedia [3]. You can also search for more information about the algorithm on the internet, of course. As a starting point for the implementation, you are provided with example code for two-dimensional arrays in C, hints on dealing with character data, and example outputs from a sequence-alignment program working on various input strings. All lecture and exercise material, the course books, and the work you did for project 1 can be helpful, too.

The deadline for handing in the source code and the report is **Friday, October 19, 2012, 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, such as sickness certified by a medical doctor, a deadline extension will be granted. Participants must notify the lecturer of such circumstances as soon as possible when they arise.

2 Starting Point

The source archive that serves as starting point for project 2 contains the following files:

- `Makefile` is the same as used for project 1.
- `test-matrix1.c` shows one way of implementing matrices in C. Here, the matrix is stored in a single array. This makes memory management easier, but requires a bit of math to access the individual matrix elements. The program also contains example code of how to nicely print a matrix, with header rows and columns and values that properly line up.
- `test-matrix2.c` illustrates an alternative method of storing matrices. Here, each row is an individual array, and the matrix is an array of pointers (one for each row). This makes memory management slightly more involved, but accessing individual matrix elements is straightforward.
- `test-propagation.c` shows one possible way of propagating values through a matrix, row by row but skipping the first row and the first column. This is similar to the propagation pattern used by the Needlemen-Wunsch algorithm, but the propagation formula is just a simplistic example.
- `test-backpointer.c` illustrates an idea for constructing backpointers and printing them in the nicely aligned table. Here, backpointers are detected by comparing whether the propagated value (called `best`) depeds on the value `above` or `left`, or both, or none. For each of

these situations, a string is constucted: it can be empty if there are no backpointers, "A" if the backpointer comes from above, "L" if it comes from the left, and "AL" if there is a backpointer from above and from the left.

- `test-argument-parsing.c` contains example code for checking whether a program receives two command-line arguments, whether those arguments contain only letters of the alphabet, and for each of their letters whether it is a vowel or a consonant. It also illustrates how to find the length of a string, and how to detect the end of a string when you are not given its length.

3 Assignment

Similarly to the first project, there are some mandatory and some bonus tasks defined for project 2. All of the mandatory tasks have to be performed, and you can choose one or more bonus tasks for additional points. A properly performed set of mandatory tasks that are well documented is worth 16 points (HH grade 4, ECTS grade D). Bonus tasks can give up to 9 extra points, such that the maximum achievable number of points is 25 (HH grade 5, ECTS grade A).

Keep in mind that writing the report is an integral part of the project, and that it also takes time. It is important to note that teams are expected to manage their resources by themselves. This includes apportioning the time available for finding information online and in the course books, developing and debugging code, documenting the work, and preparing and testing the project archive file that you submit for evaluation. How these aspects are shared between the team members is for each team to decide.

3.1 Mandatory Tasks

Create a new program called `main-sequence-align`. After completing all mandatory tasks, this program shall

1. Take two command-line arguments. Check that they contain only alphabetical letters. Create 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).

2. Follow the method described on slides 12–13 and 39–53 of lecture 6 [2] to: initialize the first row and first column of that matrix, and then propagate the costs for *insert*, *delete*, or *match* operations. The scores should be as 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 all these scores 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. Beware of the fact that matrix row i corresponds to letter $i-1$ of the source string, and matrix column j corresponds to letter $j-1$ of the destination string.

3. For each element of the matrix, determine its *backpointers*. In other words, find out (and store) which subset of the three operations (*insert*, *delete*, *match*) produced the best (minimum) value. Note that there will be at least one backpointer for every element, except for the one in the top-left corner (which gets initialized to zero and has no backpointers). *Hint: similarly to the example in `test-backpointer.c`, you can use an array of 4 characters `char[4]` and use it like a zero-terminated C string that contains 'm' for a match, 'i' for an insert, and 'd' for a delete.*

```
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 alignment scores, complete with backpointer information, for several examples of input source and destination strings. The numbers are the accumulated scores for each state, and underneath the number are up to three letters: 'm' stands for *match* (diagonal pointer), 'i' means *insert* (horizontal pointer), and 'd' means *delete* (vertical pointer).

```
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.

4. Print the contents of the matrix, along with backpointer information. An example of such output is given in figure 1 for the source "beer" and the destination "coffee" (the same example used in lecture 6) and some other examples that can be used to check your implementation.

Your report must follow the same general guidelines as for project 1. It should mention the list of tasks that you completed and describe any additional files that you have have created. It should list websites and other references that you used for your implementation. The report must also include your program's output for all of the examples used in figure 1.

4 Bonus Tasks

- After printing the table, your program should trace back an optimal alignment. For this, you start at the bottom-right element and work your way along a backpointer until you reach the top-left element. At each step of this tracing process, you have to prepend a character to an *output source* and an *output destination* string:
  - For a *match*, you use one character of each input source and destination, and proceed to the element that is diagonally up one row and left one column.

– For a *delete*, you use up one character from the input source and prepend it to the output source, but the output destination receives a gap '_'; then you proceed to the element directly one row above.

– Conversely, for an *insert*, one character from the input destination gets prepended to the output destination, whereas the output source receives a '_'; then you proceed to the element directly to the left.

Hint: the maximum length of the output strings is $N + M$, so you can allocate a buffer that is big enough (i.e. "`char *buf = calloc (N*M+1, sizeof(char))`" with one extra element for the terminating '\0') and fill it backwards starting at `buf+N*M`. You will probably not use all of the space, so skip over the first few letters when passing the resulting string to `printf`.

- 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; elements with three backpointers 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.

- 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.

5 Further Information

Preparing Archives for Submitting Your Project

You already encountered the `tar` program in exercise 7 to extract project archives. In order to create an archive, use the command "`tar xfvj archive-name.tar.bz2 project-directory`" where you have to replace `project-directory` with the name of your project directory, and the result will be in `archive-name.tar.bz2`.

However, please don't choose just any name. Rather, make a separate directory based on the family names of the team members, copy all the source and data files there, then create the archive with an archive name that has the directory name as its base.

For example, two students named "Alice Foo" and "Bob Bar" would proceed as follows:

1. Create a directory based on student and project names.
   For example: "`mkdir itads-proj2-foo-bar`"

2. Copy all the relevant files there.
   For example: "`cp *.h *.c Makefile itads-proj2-foo-bar/.`"

3. Create the archive: "`tar cfvj itads-proj2-foo-bar.tar.bz2 itads-proj2-foo-bar`"

4. **IMPORTANT!** Double-check that the files you placed in the archive are complete and free of errors. This is best done by going into the directory which you just used to create the archive, and trying to build and run your applications.

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.