Design and Analysis of Algorithms: Homework #5

Due at 11:59 PM on April 12, 2018

Professor Kasturi Varadarajan

Alic Szecsei

Problem 1

We are given two strings, called the *source* and the *target*, which both have the same number of characters n. The goal is to transform the target string into the source string using the *smallest* number of allowed operations. There are two types of allowed operations, a *flip* and a *substitution*.

- A *substitution* replaces a character in a certain position by another character. For example `eetie` is transformed to `eerie` via a substitution of the third character.
- A *flip* reverses a contiguous substring of the string. In particular, FLIP(i, j) reverses the substring starting at the ith character and ending at the jth character. For example, `eteie` is transformed to `eetie` via FLIP$(2, 3)$.

We will assume that the portions where any two flips are applied don't overlap. That is, if we perform FLIP(i, j) and FLIP(k, l), then the sets $i, i + 1, ..., j$ and $k, k + 1, ..., l$ are disjoint.

Consider the source string `timefliesslikeanarrow` and the target string `tfemiliilzejeworrbna`. The following sequence transforms the target into the source:

1. `tfemiliilzejeworrbna` (the target)
2. `tfemiliilzejeanbrrow` (a flip)
3. `tfemiliezlijeanbrrow` (a flip)
4. `timefliezlijeanbrrow` (a flip)
5. `timefliezlijeanarrow` (a substitution)
6. `timefliezlikeanarrow` (a substitution)
7. `timefliesslikeanarrow` (a substitution)

The number of operations used, which is the quantity we seek to minimize, was 6. Note that we never need more operations than the length of the given strings.

Write a time-efficient program that takes as input a file containing the two input strings and prints out the optimal sequence of transformations from the target to the source. Each string will be a sequence of lower-case English characters. Note that for the grading we plan to measure time-efficiency by actually running the program on strings of length up to five hundred or even a thousand. As background reading for designing your algorithm, it may help to study the dynamic programming approach to compute the edit distance between two strings.

Your submission for this homework should consist of two files: (a) your source code (b) a document with some text describing your algorithm and instructions on how to compile and/or run your program with a given input file. These need to be submitted in ICON as Homework 5.

Solution

The algorithm runs in $O(n^3)$ time. We work backwards from the target to reach the source, assuming 1 fewer letter has been edited each time.

Thus, we have two options: either the current letter matches the target letter, or it does not. If they match, we simply use the amount of time it took to transform everything afterward; if they do not match, we have several options.

The first is a simple substitution. We can replace the value of the source with the value of the target; this adds one transformation to our history, and does not affect any other transformations, so we can then refer back to the edit

distance for the remainder of the source.

Alternately, we could perform a flip. Here, we are not certain how large a flip would be optimal, but we do know that it is impossible to flip the same section twice. Our flip can therefore *also* add transformations to substitute any mismatched letters enclosed by the flip, without worry that we may be skipping potential flipping transformations. Using this, we can attempt to flip all possible numbers of characters, and determine how effective that particular flip is.

For whichever of these transformations causes the smallest resultant transformation history, we memoize this history in an array, and progress backwards until eventually we can determine, from the very first character, which course of action is most efficient.

The main body of our algorithm runs in $O(n^2)$ time; we have a clear outer loop that iterates our actionable character from the end of the target back to the beginning. Each time it does so, we must iterate over all possible swaps that may be performed from that actionable character, which combine to form $O(n^2)$ runtime. However, actually performing the swap requires $O(n)$ operations; this must be performed for every swap, giving us a combined running time of $O(n^3)$.

When testing, using maximal optimization compiler flags, the program was able to find the edit distance of 1000-character strings in approximately 12 minutes; 500-character strings took just over 1 minute.

To run the program, download the zipped project file using Visual Studio and build it. Visual C++ must be installed to do this.

Using "Release" mode, build the solution. To run the built executable, in the terminal type:

```
AlgorithmsHW5.exe [inputfile] [outputfile?]
```

The `[outputfile?]` argument is optional, and is included for larger test cases where the console is not sufficient to view a long list of transformation operations.