CS358: Applied Algorithms

Assignment 2: Space-Efficient Edit Distance (due 10/2/25)

Instructor: Sam McCauley

Instructions

There are two parts to the assignment: code, and a pdf submission.

Code can be submitted by committing and pushing them to gitlab. I strongly suggest that you access `evolene.cs.williams.edu` through a web browser to make sure everything was uploaded as you expected. You may collaborate on code with your classmates, as well as the instructor and TA, but you may not use any LLM assistance.

The pdf should be uploaded to Gradescope for submission and feedback. The pdf should be entirely your own work; you should only use "hands in pockets" discussions with your classmates and the TA; you should not use any online (including LLM) assistance.

Please contact me at srm2@williams.edu if you have any questions or find any problems with the assignment materials.

Problem Description

Input: The input consists of a sequence of tests. Each test begins with a line that has four numbers on it. These numbers are the length of `string1`, the length of `string2`, the length of the intended solution string, and finally the size of the alphabet. Following this line, `string1` is listed in 80-character lines, followed by `string2` and finally the intended solution. These strings each have brief comments (to make it more human-readable) that are ignored by the input reader provided to you.

Output: The output is a string describing a sequence of edits. Each character in the string should be `'i'`, `'r'`, `'d'`, or `'m'`. The string should be null-terminated.

Goal: The output is a string describing how `string2` can be modified to obtain `string1`. The output is a string of characters, where each character is `'i'`, `'r'`, `'d'`, or `'m'`, representing an insert, replacement, delete, or match, respectively.

Thus, if `string2` is `ac`, and `string1` is `a`, the optimal output string is `md`.

There is a function in `test.c` that will make sure your sequence of edits is the right length, and ensure that it correctly edits `string2` to obtain `string1`. There may be many correct answers; any of them should be recognized as correct by this function.

Testing Parameters

The `main()` method of the testing program (in `test.c`) takes one argument, which is a file containing edit distance instances. You can test your program by first running `make`, and

then running (say) `./test.out smallData1.txt`. As always, you should first debug on the small instances before testing on the larger instances.

There are four files given. `smallData1.txt` is very small, and useful for debugging. `smallData2.txt` contains further small instances, and is particularly useful for testing base cases. Note that a correct answer is given for each instance, which may be useful for debugging. `data1.txt` and `data2.txt` contain larger instances, each taken from either human DNA, or from English text.

`test.out` takes an extra optional parameter. If this parameter is given, then all strings are set to be exactly the given length (they are either truncated to this length, or duplicated until reaching that length). For example, `./test.out data2.txt 30000` will set all strings to length 30000, and then find the edit distance between them.

Helper Functions

The following functions are in `helper.c` and are there to help you in your implementation. You may modify them if you wish. See the comments in the code for more detail on how to call them.

- `minOfThree`: returns the min of three `int`s.
- `reverseString`: returns the reverse of a string
- `baseCase()`: base case implementation for Hirshberg's algorithm. Works if at least one of the strings has length at most 1.
- `editCostRow()`: gives the cost of the last row of the dynamic programming table between two given strings. Can be used to find the "crossover" point for Hirshberg's algorithm.
- `backtracking()`: the backtracking implementation of the edit distance seen in class. This requires much more space than Hirshberg's algorithm.

Questions

Implementation 1. Implement Hirschberg's space-efficient algorithm for edit distance. Change the code so that `editDistance()` in `editDistance.c` calls your implementation (see the comments in the code for a few small hints).

Implementation 2. Compare the running time of your Hirshberg's implementation with the running time of the space-inefficient function `backtracking()` (recall that this is given to you in `helper.c`). You should compare the times for strings of size 20k, 40k, 60k, 80k on data set `data2.txt`. Your results should be in the form of a plot.

You should also include the time for 100k for your Hirshberg's implementation; the

space-inefficient version will likely be killed by the operating system due to using too much memory (you should test this and see if it's the case).

Solution. □

Implementation 3. Use `cachegrind` to compare the (simulated) number of cache misses of your Hirshberg's implementation with the cache misses of the space-inefficient function `backtracking()` on the data set `data2.txt`.

You do not need to make a plot or change the length of the strings; just give the number of L1 and LL cache misses below.

Solution. □

Shelving Books with Labels

Updated 9/25

Let's say you work in a library. You have to put m books (let's call them $b_1, \ldots, b_m$) on $n \leq m$ shelves (which we'll call $s_1, \ldots s_n$). The books are numbered using the Dewey Decimal system, and must be placed in order starting on shelf s_1. Furthermore, there may not be an empty shelf between two shelves that contain books.[1] For example, if book 10 goes on shelf 2, book 11 must go on shelf 2 or shelf 3. Each shelf can hold any number of books; even all m books may be placed on a single shelf. The books must start being placed on the first shelf.

This would normally be fairly easy—for example, you could just put all books on the first shelf. Unfortunately, this library also keeps track of k *topics* to help people browse for books they may be interested in. Each shelf s_j has a label ℓ_j representing the topics of books on that shelf. Similarly, each book b_i has a list of topics t_i representing what topics are covered in that book.

You were instructed to reprint all the labels on the shelves so that the shelves indicate the topics of their books: if book b_i is on shelf s_j, then each topic in t_i can be found in ℓ_j. However, in an effort to stay green you want to keep the labels as-is, and place the books so that they match the current labels as closely as possible (while still retaining Dewey Decimal order).

Let's say that the *cost* of placing book b_i on shelf s_j is the number of topics t_i that do not appear in the list ℓ_j. This leads to an algorithmic problem: how can the books be assigned to shelves to minimize the total cost; i.e. the number of missing topics over all books on all shelves?

Dynamic Program. The above book shelving problem can be solved with the following dynamic programming algorithm.

Firs, the **subproblems**. We will consider the problem of putting the first i books on the first j shelves (where **all** j shelves are used; so the ith book is on the jth shelf). We will

[1] Your boss at the library is a stickler for aesthetics.

store this cost in a table $T[j][i]$. Since we require that the first and last shelf are used, and that there are no gaps, we only consider entries where $i \geq j$ (since if $i < j$ you would not have enough books to cover all the shelves).

This table is sometimes called the **memoization data structure**. We will fill out the table for $i = 1 \ldots m$ and $j = 1 \ldots n$, so it is an $n \times m$ table (that is to say: a table with n rows and m columns).

Now, let's look at the **base cases**. If $n = 1$, all books must be stored on one shelf, and the total cost is the sum of the costs of putting all books on the first shelf. Otherwise, if $n = m$, then the number of books is equal to the number of shelves. This means the ith book must be on the ith shelf, summing over the books/shelves gives us the cost.

Now, the heart of the dynamic program: the **recurrence**. We want to find a recurrence for $T[j][i]$: the cost of putting the first i books on the first j shelves. We know that book i must be stored on shelf j. There are two possibilities for book i: either it is the first book on shelf j, or it is not. If it is the first book on its shelf, the cost is the cost of putting all previous books on all previous shelves (which we have calculated as $T[j-1][i-1]$), plus the cost of putting book i on shelf j. If it is not the first book on its shelf, it goes on shelf j, and the previous books must also be on shelves 1 through j (which we have calculated as cost $T[j][i-1]$). We take the minimum between these options. That gives the following equation (which denotes the cost of putting book i on shelf j as c_{ij} for simplicity):

$$T[j][i] = \min\{T[j-1][i-1] + c_{ij}, T[j][i-1] + c_{ij}\}$$

We fill in the table row by row. Recall that in the base case, we filled in $T[1][1]$ through $T[1][n]$, and we also filled in $T[j][j]$ for all j (and recall that we assume that $i \geq j$). We begin our recursion on the second row: $T[2][2]$ is already filled in, so we begin by filling in $T[2][3]$, then $T[2][4]$, and so on until $T[2][n]$. Then we go to the next row: $T[3][3]$ is filled, so we fill $T[3][4]$ then $T[3][5]$, and so on. The **final answer** is stored in $\min_{j \leq m} T[j][n]$.

Problem 1. Analyze the above dynamic programming algorithm in the external memory model—how many cache misses does it have in terms of n, m, and B?

You should assume for this analysis that n and m are much larger than B. Furthermore, assume the table is stored in row-major order: in a single cache miss we can bring read or write (say) $T[1][1]$ through $T[1][B]$.

Solution. □

Problem 2. If the table T was stored in **column-major** order, how many cache misses would it have? (You only need to give the bound and a 1-sentence explanation.)

Solution. □

As we saw with edit distance, if we only care about the *cost* of a solution (rather than reconstructing the solution itself), there's an immediate optimization that saves space.

Problem 3. Give an algorithm to find the minimum **number** of mismatches in $O(nmk)$ time and $O(n+m)$ space (you do not need to find the optimal assignment of books to shelves).[a] A one-to-two sentence explanation is enough.

[a] You may notice that it's probably possible to tighten the space a bit, to something like $O(\min\{n, m\})$. This is not required.

Solution. □

Finally, let's use the ideas from Hirschberg's algorithm to improve the space usage while giving the assignment itself, not just the cost.

Problem 4. Finally, give an algorithm to find the assignment of books to shelves that minimizes the number of mismatches in $O(nmk)$ time and $O(n+m)$ space.

You will probably want to solve this in two parts. First, in $O(nmk)$ time and $O(n+m)$ space, find the optimal last shelf j'.[a] Then, we can assume that the last book is placed on shelf j'; with that assumption, we are ready to use a Hirschberg's-like strategy.

[a] This first step is important in order to use the "reversing" trick to determine the optimal splitting point—you should make sure you understand why that is.

Solution. □