CS358: Applied Algorithms

Homework 2: Space-Efficient Edit Distance (due 9/26/24)

Instructor: Sam McCauley

Instructions

All submissions are to be done through github. This process is detailed in the handout "Handing In Assignments" on the course website. Answers to the questions below should be submitted by editing this document. All places where you are expected to fill in solution are marked in comments with "FILL IN."

Please contact me at sam@cs.williams.edu if you have any questions or find any problems with the homework 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`.

Please pay attention to ties. Your algorithm should favor going as far right in the dynamic programming table as possible, if `string1` is represented vertically and `string2` is represented horizontally.

- This means that when implementing the recursive algorithm, if there are multiple splits that have the same cost, you should take the rightmost option.
- Equivalently, if you are implementing a non-recursive algorithm, if there is a tie while backtracking you should use an insert whenever possible, then a match or replace, and finally a delete.
- If your algorithm breaks ties correctly, if `string1` is `aaa` and `string2` is `aa`, it should output `mmi` (not `mim` or `imm`). The input file `testBaseCases.txt` has more examples to help you ensure that your algorithm breaks ties correctly.

Testing Parameters

The `main()` method of the testing program (in `test.c`) takes two arguments, each of which is a file containing edit distance instances. You can test your program[1] by first running `make`, and then running `./test.out testData.txt timeData.txt`.

- All instances on this homework will have an alphabet of size 4 or 256; in either case the input will be taken as a normal array of `char`s. You may optimize your solution under these assumption (so it's OK if your algorithm fails with alphabets that are of any size other than 4 or 256).
- There are several testing and timing instances provided. Your performance will be judged based on the total time across three instances. The three testing instances will look very similar to those provided in `timeData.txt`.
- All 4-character instances have `string1` as an actual subsequence of human DNA, and `string2` as a perturbed version of this DNA. All 256-character instances have `string1` consist of English text (including punctuation), and `string2` as a perturbed version of this text.
- Tiebreaking is unfortunately necessary and unavoidable even in the large instances.
- Two solutions with running times within .1 seconds of each other will be considered tied for the purposes of this homework.

Questions

Code (50 points). Implement the standard dynamic program for edit distance. Then, implement Hirschberg's space-efficient algorithm. Please give a very brief (2-3 sentence) description of your implementation below.

Solution.

Problem 1 (10 points)**.** Use `cachegrind` to compare the performance of your space-inefficient approach, and your approach using Hirschberg's algorithm. Interpret the data: which has more (data) cache misses? What level of cache is incurring those misses?

Describe how you performed your tests, and include the output of your tests.

Solution.

[1]While you're debugging, you may instead want to test base cases rather than the big input: run `make`, then `./test.out testData.txt testBaseCases.txt`

External Memory

Problem 2 (**Extra credit**: 15pts)**.** Give an algorithm that can find the edit distance (not necessarily the sequence of edits) between strings of length n and length m in $O(nm/MB + (n+m)/B)$ I/Os. (Your algorithm should work even for very large m and n.) Analyze your algorithm, explaining why it meets this bound.

Solution.

Shelving Books with Labels

Let's say you work in a library. You have to put m books (let's call them $b_1, \dots, b_m$) on $n \leq m$ shelves (which we'll call $s_1, \dots s_n$). 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.[2] 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.

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 store this cost in a table $T[j][i]$.

This table is sometimes called the **memoization data structure**. We will fill out the table for $i = 0 \dots m$ and[3] $j = 1 \dots n$, so it is an $n \times (m+1)$ table (that is to say: a table with n rows and $m+1$ 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.

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

[3]We start j at 1 since the problem does not make sense to me with 0 shelves.

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):

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

We fill in the table row by row: we fill in $T[1][0]$, then $T[1][1]$, then $T[1][2]$ and so on; after $T[1][m]$ we fill in $T[2][0]$, and so on. The **final answer** is stored in $T[n][m]$.

Problem 3 (10 points)**.** 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) $T[0][0]$ through $T[0][B-1]$.

Solution.

Problem 4 (5 points)**.** 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 5 (15 points)**.** 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).[4] A one-to-two sentence explanation is enough.

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 6 (10 points)**.** 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.

Solution.

[4]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.

Tips and tricks

- Remember to create a correct program before worrying about creating a fast one! Even more importantly: create a correct program before creating one that cleverly reuses unnecessary space. Even your first Hirschberg's implementation should probably be fairly wasteful!

- I would fairly strongly suggest that you create a working version of the simple (not space-efficient) edit distance algorithm to help you with debugging.

- Keep an eye on memory management! If you are not careful you may wind up with $\Theta(nm)$ memory usage even with a recursive algorithm.

- As I mentioned in class, a correct implementation of Hirschberg's algorithm almost certainly has disjoint (nonoverlapping) subproblems.

- In class we discussed that maintaining the solution using Hirschberg's algorithm can be a bit tricky. I believe that the easiest way is to do it "bottom-up:" construct a solution in the base case, and pass it to the calling function. The calling function can then allocate space for a solution that combines its two recursive calls, and (again) pass it up to its calling function; and so on.

- As a reminder, valgrind is an excellent tool if you are having trouble keeping track of memory. It is very easy to use, and it is available on the lab computers.

- While it is likely possible to implement an $O(\min\{m, n\})$ space algorithm, it is probably better to implement an $O(n + m)$ space algorithm and then work on other avenues of improving efficiency.