CS 4349.003.19F Lecture 9—September 18, 2019

Main topics for #lecture include #dynamic_programming including #example/edit_distance.

Announcements

- Midterm 1 will be held in class on Wednesday, October 2nd. If you have a conflict, please let me know ASAP. I'm doing this as a closed book, no notes or calculators exam.
- It will contain a combination of questions designed to check your understanding of the past lectures as well as some algorithm design questions.
- The topics will include everything up to and including dynamic programming. So running time analysis, recursion and induction, divide-and-conquer, and dynamic programming.
- Greedy algorithms ARE NOT covered on Midterm 1.
- Also, our TA Joe will be out of town for a conference the next few days. He normally holds office hours on Fridays so I'll hold my own from 2 to 3 this Friday, September 20th. Please come! I'm more than happy to walk through homework problems or discuss things from class, and students that show up really do perform better on both homework and the exams.

Edit Distance

- We're going to continue with dynamic programming today by looking at a way to compare pairs of strings.
- The *edit distance* between two strings is the minimum number of character insertions, deletions, and substitutions required to transform one string into the other.
- For example, the edit distance between FOOD and MONEY is at most four:

```
FOOD → MOOD → MON D → MONED → MONEY
                ^
```

- This distance function was independently proposed by Vladimir Levenshtein (working in coding theory), Taras Vintsyuk (working on speech recognition), and Stanislaw Ulam (working on biological sequences), so it's often referred to as Levenshtein or Ulam distance (but never Vintsyuk distance for some reason).
- The way I defined the problem lets us do edits in any order, but adding some organization will help us design a dynamic programming algorithm.
- One way to add some organization is to ask what happens to each character of each string.
- So let's visualize the editing process by aligning the two strings on top of one another to represents what happens to a character of the first string or where the character of the second string comes from.

```
F O O   D
M O N E Y
```

- There's a gap in the first word for every insertion and a gap in the second word for every deletion. If a column contains two *different* characters then it represents a substitution.
- Another example, ALGORITHM vs ALTRUISTIC. The edit distance is at most 6.

```
A L G O R   I   T H M
A L   T R U I S T I C
```

- But how do we compute the edit distance exactly? Let's design a dynamic programming algorithm that, given two strings A[1 .. m] and B[1 .. n], returns the edit distance between A and B.

Recursive Structure

- The first step in any dynamic programming algorithm is developing a recursive algorithm or recurrence.
- This alignment representation suggests a recursive structure.
- **If we remove the last column from the optimal alignment structure, then the remaining columns must represent the shortest edit sequence for the remaining prefixes.**
- There's an easy proof by contradiction here. If the prefixes had a shorter edit sequence, then gluing the last column onto the shorter sequence would give us a better sequence for the original strings than we started with.
- So, we should guess what to do with the last column and have the Recursion Fairy take care of the rest.
- Here’s another way to think about it.
- The alignment we're looking for represents a sequence of editing decisions ordered (for convinience of notation later) from right to left.
- We need to make a sequence of decisions, one for each column of the alignment.
- In the middle of these decisions, we have already aligned a suffix of one string with a suffix of the other.

ALGOR	I		T	H	M
ALTRU	I	S	T	I	C

- The total cost of the alignment is just the number of mismatched columns, so our remaining decisions don't depend on the editing operations we've already chosen. They only depend on the prefixes we haven't aligned yet. The Recursion Fairy only needs to care about the two prefixes.

ALGOR
ALTRU

- So let's describe the problem we're giving to the Recursion Fairy using a recurrence. Let Edit(i, j) denote the edit distance between A[1 .. i] and B[1 .. j]. We ultimately need to

compute Edit(m, n).

Recurrence

- When i and j are both positive, there are three possibilities for the last column.
  - Insertion: The last entry of the top row is empty. Here, the edit distance is equal to 1 + Edit(i, j - 1). The 1 is the "cost" of doing one insertion and the recursive call is us having to handle all i characters of A[1 .. i] still but only the first j - 1 characters of B[1 .. j].

  - Deletion: The last entry of the bottom row is empty. Here, the edit distance is equal to 1 + Edit(i - 1, j). Here, we have all of B[1 .. j] left, but we already took care of the last character of A[1 .. i].

  - Substitution(?): Both rows have characters in the last column. If the characters are different, the edit distance is 1 + Edit(i - 1, j - 1). Otherwise, the “substitution” is free so the edit distance is Edit(i - 1, j - 1).

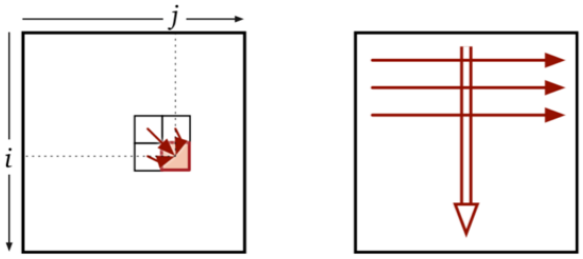
- This case analysis doesn’t work when one of i or j is equal to 0, but we can handle the boundary cases easily.
  - Converting an empty string into a string of length j requires j insertions, so Edit(0, j) = j.
  - Converting a string of length i into the empty string requires i deletions, so Edit(i, 0) = i.
- And either of those rules imply the edit distance of two empty strings Edit(0, 0) = 0. Checks out.
- So the Edit function follows this recurrence.

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \begin{Bmatrix} Edit(i,j-1)+1 \\ Edit(i-1,j)+1 \\ Edit(i-1,j-1)+[A[i] \neq B[j]] \end{Bmatrix} & \text{otherwise} \end{cases}$$

- Here, I’m using the notation [P] that returns 1 if proposition P is true and 0 otherwise.

Dynamic Programming

- And now we can find our efficient algorithm using the mechanical memoization process.
  - Subproblems: Each recursive subproblem takes a pair of indices $0 \leq i \leq m$ and $0 \leq j \leq n$.
  - Memoization: So we can memoize all possible values of Edit(i, j) in a two-dimensional array Edit[0 .. m, 0 .. n].
  - Dependencies: Each entry Edit[i, j] depends only on its three neighboring entires Edit[i, j - 1], Edit[i - 1, j], and Edit[i - 1, j - 1]. Here's a picture of what it looks like.



  - Evaluation order: We only need things from earlier in the same row or from the previous row, so let's fill the array in row-major order: row by row from top down, each row from left to right.
  - Space and time: At this point, we can already figure out the running time without writing any code! The memoization structure uses **O(mn) space**. We can compute each entry Edit(i, j) in O(1) time once we know its predecessors, so computing all the entries will take **O(mn) time**.
- And finally, here's the algorithm!

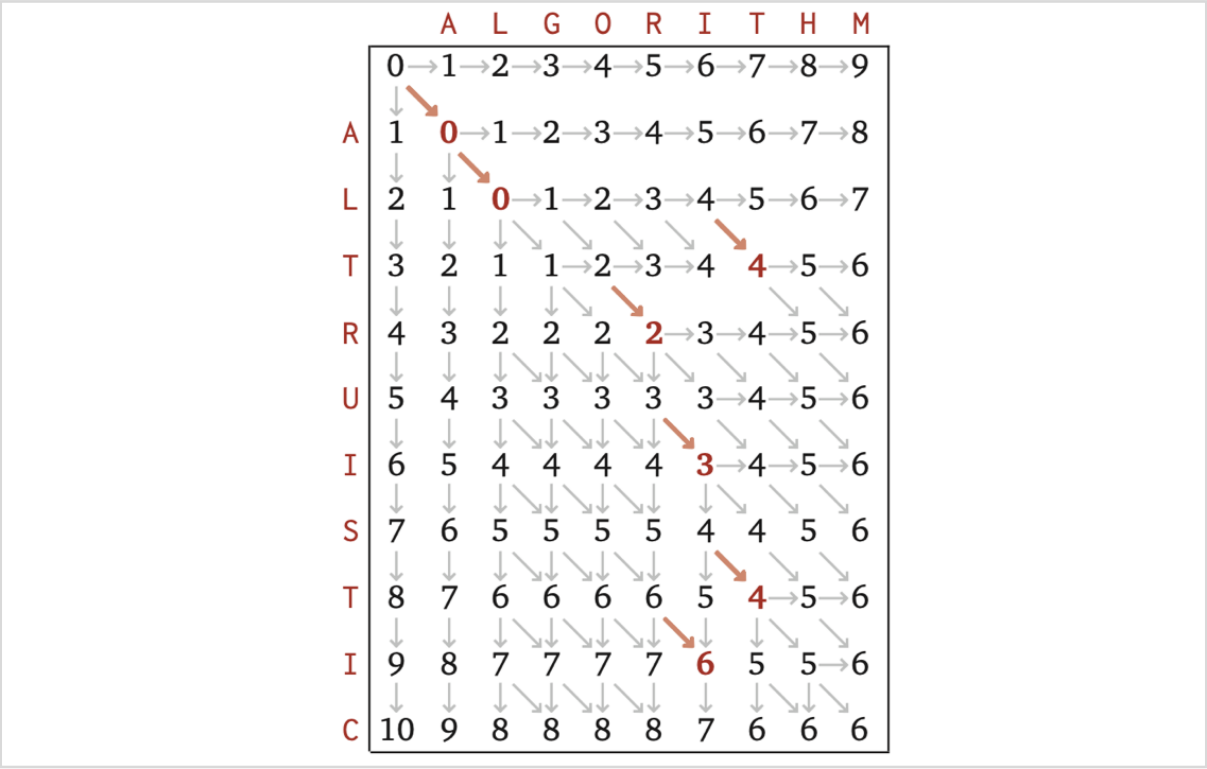
```
EditDistance(A[1..m], B[1..n]):
  for j ← 0 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          ins ← Edit[i, j − 1] + 1
          del ← Edit[i − 1, j] + 1
          if A[i] = B[j]
              rep ← Edit[i − 1, j − 1]
          else
              rep ← Edit[i − 1, j − 1] + 1
          Edit[i, j] ← min {ins, del, rep}

  return Edit[m, n]
```

- We started by finding the recursive structure and eventually got to a relatively simple iterative process.
- A brief historical note: Most people attribute this algorithm to Wagner and Fischer who described it in 1974. But it was found by several other at the same time or even a bit earlier.

- Now, similar to how we have recursion trees to explain what's going on with our recursive algorithms, we can also look at how the array is filled. I want to emphasize again, though, that **designing the recursive algorithm was the key step**. We're only looking at the array to see what happens in hindsight.



- Suppose we're transforming ALTRUISTIC into ALGORITHM.
- A number at position (i, j) is the value of Edit(i, j). The arrows indicate which predecessors could have defined each entry. A horizontal arrow means we did an insertion. A vertical arrow represents deletion, and a diagonal arrow represents substitution. Bold arrows are the "free" substitutions that don't increase the cost.
- The algorithm we wrote only computes the total edit distance, not the optimal sequence of operations. However, any path of arrows from (0, 0) to (m, n) represents an optimal sequence of operations.
- If we want to recover an optimal sequence, we can figure out which arrows lead into any entry (i, j) in O(1) time by checking its three dependences and see which one gives us the correct cost. In O(m + n) additional time over what it takes to compute the edit distance, we can reconstruct an optimal sequence by tracing *backwards* from (m, n).
- This is one example of a trend. If you want to find an optimal solution for some problem using dynamic programming, its easiest to first write an algorithm to find the *value* of the solution and then make any changes necessary to get the solution using the values.

Common Patterns

- The two examples we've seen so far of string segmentation and edit distance follow some common patterns.

- We're given one or more sequences (arrays) as input, and our goal is to compute an optimal sequence as output: a sequence of word boundaries or a sequence of editing operations, for example.
- Usually, the input to recursive subproblems in these cases consists of *prefixes* or *suffixes* of the input arrays, possibly with additional information.
- For example, if I wanted to use at most k substitutions, then I'd use a third parameter telling me how many substitutions I have remaining in my recursive subproblem. For string segmentations I could limit the number of words we use by passing along an additional parameter.