Project 2 Report

Kenneth Sills, Kevin Orr, Elijah Malaby

July 16, 2017

1 Breaking Down Problems

Insertion, deletion, and substitution can be solved using a simple recurrence that can be easily memoized. Transposition proved to be more involved. We separate the transpositions from all other typo forms. By first keeping a record of where possible transpositions can occur (and each permutation of possible propagation lengths), we vastly simplify the recurrence. In the memoized solution, we also separate the initial table fill from backtracking.

2 Parameters for the recursion

The two arrays p and t (pattern and typo) and indexes i and j representing the positions in p and t currently being compared

3 What recurrence can you use

The naive recurrence we came up with was:

$$C(p,t) = \begin{cases} \min \left\{ \begin{array}{l} C(p[1..m-1], t) + insertCost(p[m], p[m-1]), \\ C(p, t[1..n-1]) + deleteCost(t[n], t[n-1]), \\ C(p[1..m-1], t[1..n-1]) + substituteCost(p[m], t[n]), \\ C(p[1..m-1], t[1..n-2] \,||^{1}\, t[n]) + transposeCost(t[n-1], t[n]) \end{array} \right\}, & m > 1, n > 1 \\ C(p[1..m-1], t) + insertCost(p[m], p[m-1]), & m = 1, n > 1 \\ C(p, t[1..n-1]) + deleteCost(t[n], t[n-1]), & m > 1, n = 1 \\ 0, & m = 1, n = 1 \end{cases}$$

$$\text{where m} = \text{length(pattern)}, \quad n = \text{length(typo)}$$

This ended up being extremely inefficient in practice: the transposition operator doesn't allow us to memoize the main function using simply a 2d array, since transposing affects the typo string when recursing. We can't simply assume that the typo string and pattern strings are simply truncated between calls.

4 What are the base cases

Whenever i or j are equal to 1. If $i = 1$, the remaining characters in $t[1..j-1]$ were trivially all insertions at the beginning. If $j = 1$, the remaining characters in $p[1..i-1]$ were deleted. If both $i = 1$ and $j = 1$, there are no further characters to compare.

[1] Note that foo is the string concatenation operator here

5 What data structure would you use

A map from pairs of (i, j) to the cost of the recurrence for (i, j), as well as a map from pairs of (i, j) to a set of numbers of transpositions k such that for each a chain of transpositions may have occurred in $p[(i-k)..i]$ and $t[(j-k)..j]$.

We found that the best structure to use for memoizing would be 2D array with each cell containing the lowest cost typo, the overall cost of accumulated typos, and the index to the lowest cost next cell.

As well, a hash table is kept, indexed by the squashed coordinates of our 2D array to store possible transposition sites. This hash table contains arrays of size 12, where the distance into the table is the number of propagating transpositions, the contents of which is the cost to perform that chain of transpositions.

6 Pseudocode for a memoized dynamic programming solution

```
Input: data: Table containing the memoized data
Input: transposes: Set of possible transpositions
Input: correct: The correct string
Input: actual: The actual string with typos
Input: i: Current position into the correct string
Input: j: Current position into the actual string
Output: Running cost of typos
Algorithm Main ():
    Fill the parents of data with (−1, −1)
    find_transposes ()
    Fill (i, j)
    Let p be (i, j)
    until p == (−1, −1):
        Record the error made at data[p]
        p = data[p]
    return the recorded errors
```

```
def find_transposes ():
    for i from length (correct) to 2:
        for j from length (actual) to 2:
            Let correct_char be correct[i]
            Let current_char be actual[j]
            Let left_char be actual[j − 1]
            if left_char == correct_char and current_char != correct_char:
                Start the running cost at transpose_cost(left_char, current_char)
                for k from 1 to 12:
                    Let correct_char be correct[i − n]
                    Let left_char be actual[j − 1 − n]
                    if current_char == correct_char:
                        Add a possible transposition to transpoes[i, j] with the current running cost
                    if correct_char != left_char:
                        break the innermost loop
                    Add transpose_cost(left_char, current_char) to the running cost
```

```
def Fill(i, j):
    if data[i, j] has a value:
        return the cost in data[i, j]
    elif i == 1 and j == 1:
        return 0
    elif i == 1:
        Let the cost be insert_cost(i, j) + Fill(i, j − 1)
        Store the cost in data[i, j]
        Set the typo of data[i, j] to Insert
        Set the parent of data[i, j] to (i, j − 1)
        return the cost
    elif j == 1:
        Let the cost be delete_cost(i, j) + Fill(i − 1, j)
        Store the cost in data[i, j]
        Set the typo of data[i, j] to Delete
        Set the parent of data[i, j] to (i − 1, j)
        return the cost
    else:
        Let options be a list of possible errors.
        Add an Insert error to options with cost insert_cost(i, j) + Fill(i, j − 1) and parent (i, j − 1)
        Add a Delete error to options with cost delete_cost(i, j) + Fill(i − 1, j) and parent (i − 1, j)
        if correct[i] == actual[j]:
            Add a None error with cost Fill(i − 1, j − 1) and parent (i − 1, j − 1)
        else:
            Add a Substitute error to options with cost substitute_cost(i, j) + Fill(i − 1, j − 1) and
              parent (i − 1, j − 1)
        if There are transpositions in transpoes[i, j]:
            for t in transpoes[i, j]do
                Add a Transpose error with cost t.cost + Fill(i − t.length, j − t.length) and parent
                  (i − t.length, j − t.length)
        Pick the minimum option in options
        Store the cost, error type, and parent in in data[i, j]
        return the cost
```

7 Worst case time complexity

The worst-case of our algorithm will encompass:

- O(n * m) Transposition identification.
- O(n * m) Fill table pass
- O(n + m) Backtracking pass

So, as a whole, the algorithm takes O(n * m) time complexity.

8 Pseudocode for nested loop

9 Can the space complexity of the iterative algorithm be improved relative to the memoized algorithm

No, because the iterative algorithm has to build up every case from the base cases regardless of if that case will be used to compute the final result. The memoized algorithm, on the other hand, will only used enough space to store the results of the cases used.

10 Describe one advantage and disadvantage of the iterative algorithm