October 19, 2017

We quickly present several different ways to solve edit-distance. This note is intended to demonstrate the different ways to do memoization, and how to get a dynamic program in the end of the process.

1 The different ways to remember

1.1 Problem and recursive formula

Definition: Given two strings $X = x_1 x_2 \ldots x_n$ and $Y = y_1 \ldots y_n$, their edit distance is the minimum cost of edit operations that covers the string X to the string Y. Here, deleting or inserting a characters costs δ, and changing a letter u to a letter v costs $\alpha(u, v) \geq 0$ (here, $\alpha(u, u) = 0$, for all u).

Given X and Y as global variable, let $f(i, j)$ be the cost of the optimal edit distance between the prefix $x_1 \ldots x_i$ and $y_1 \ldots y_j$. It is not hard to see that this yield the following recursive function (see class notes):

$$f(i,j) = \begin{cases} \delta i & j = 0 \\ \delta j & i = 0 \\ \min \begin{cases} \alpha(x_i, y_j) + f(i-1, j-1) \\ \delta + f(i-1, j) \\ \delta + f(i, j-1) \end{cases} & \text{otherwise.} \end{cases}$$

We are interested in computing $f(n, m)$.

1.2 Recursive code

The input is provided in two global strings $X[1 \,..\, n$ and $Y[1 \,..\, m]$. We are interested in computing **editDistRV**(n, m).

```
editDistRV(i, j):
  if j = 0 then return δ * i
  if i = 0 then return δ * j
  v1 ← α(X[i], Y[j]) + editDistRV(i − 1, j − 1)
  v2 ← δ + editDistRV(i − 1, j)
  v3 ← δ + editDistRV(i, j − 1)
  return min(v1, v2, v3).
```

1.3 Edit distance with implicit memoization

We first initialize a lookup data-structure m that is initially empty (it can be implemented using a hash-table, or a map). When you lookup a value that is not already stored in m, it returned that this value is not defined. Here, we are interested in computing **edM**(n, m).

```
edM(i, j):
  if j = 0 then return δ * i
  if i = 0 then return δ * j
  if m((i, j)) is defined then
      return m(i, j).
  v1 ← α(X[i], Y[j]) + edM(i − 1, j − 1)
  v2 ← δ + edM(i − 1, j)
  v3 ← δ + edM(i, j − 1)
  v ← min(v1, v2, v3).
  Store the value v in m, with the key is (i, j).
  return v
```

1.4 Edit distance with partial memoization

One can use a table or an array instead of a map data-structure to remember the arrays.

```
Input: X[1 .. n and Y[1 .. m]
Init:
  for i = 0 to n do
      for j = 0 to m do
          M[i][j] ← ∞

Main:
  return edP(n, m):

edP(i, j):
  if j = 0 then return δ * i
  if i = 0 then return δ * j
  if M[i][j] < ∞ then
      return M[i][j].
  v1 ← α(X[i], Y[j]) + edP(i − 1, j − 1)
  v2 ← δ + edP(i − 1, j)
  v3 ← δ + edP(i, j − 1)
  M[i][j] ← min(v1, v2, v3). return M[i][j]
```

Note, that this would be significantly faster in practice than implicit memoization – you are avoiding the overhead of the map and the memory management associated with it, which could be quite significant.

1.5 Dynamic program for edit distance: Explicit memoization

With explicit memoization you just fill the table directly, and you skip the recursion all together – you just need to be very careful how you fill the table. This is going to be even faster, but the speedup is probably going to be relatively small (it might be significant if the function is truely simple, as in this case.

```
edDP( X[1 .. n], Y[1 .. m] ):
  Allocate table M[n][m]
  for i = 0 to n do
      M[i][0] ← i * δ
  for j = 0 to m do
      M[0][j] ← j * δ

  for i = 1 to n do
      for j = 1 to m do
          v1 ← α(X[i], Y[j]) + M[i − 1][j − 1]
          v2 ← δ + M[i − 1][j]
          v3 ← δ + M[i][j − 1]
          M[i][j] ← min(v1, v2, v3).

  return M[n][m]
```

You should always shoot for having a solution using **explicit memoization** (i.e., a **dynamic program** solution) – it usually results in faster (and in many cases simpler code). The idea of going through recursion to implicit memoization is to help you understand the problem enough so that you can generated a dynamic program for it.

1.6 Dynamic program with less space

In the edit distance case, it is easy to reduce the space requirement from $\Theta(nm)$ to $O(n)$, by observing that you need to remember only two columns of the table as you fill it. Here is the resulting dynamic program. See slides for more details.

```
editDistanceLS( X[1 .. n], Y[1 .. m] ):
    for i = 1 to n do P[i] ← iδ
    for j = 1 to m do
        C[0] ← jδ (* corresponds to M(0, j) *)
        for i = 1 to n do
            v1 ← α(X[i], Y[j]) + P[i − 1]
            v2 ← δ + C[i − 1]
            v3 ← δ + P[i]
            C[i] ←= min(v1, v2, v3)
        for i = 1 to n do
            P[i] ← C[i]

    return P[n].
```

1.7 Backtracking?

Backtracking is irrelevant to dynamic programming. Forget you ever heard this word. It was a dream.