CSCI270 Week 7

Jacob Ma

October 18, 2022

1 Edit Distance / Sequence Alignment (Dynamic Programming)

Problem 1

Given two strings $x[1, \cdots, n]$, $y[1, \cdots, m]$, how similar are they?

Different notions of similarity:

(1) Hamming Distance: Number of characters that different between $x[i]$ and $y[i]$. Not used here.

(2) Edit Distance: Cost for inserting characters, cost for deleting characters, cost for replacing characters.

Edit Distance between x and y is the minimum cost of any sequence of operations turning x into y.

Here, for specificity, assume inserting/deleting both cost A, overwriting costs B.
We can easily generalize everything to different costs for insertion/deletion, and even to character-specific insertion/deletion/replacement costs.

Alternative way to think of deletions/insertions/replacements: *Aligning Strings*.

$$ababbcde\_$$
$$\_\_abcc\_ef$$

Insert blanks into both strings wherever we want, to make them the same length. Then pay A whenever a character is aligned against a blank, and B whenever a character is aligned against a different character.
Goal: find blank insertions that minimize the total cost.
These two problems are equivalent: blank insertions into y correspond to deletions from x; blank insertions into x correspond to insertions in going from x to y; and mismatched characters correspond to replacements.

What do we know for sure about the optimal alignment of $x[1 \cdots n]$ with $y[1 \cdots m]$?
The last position of the optimal alignment contains one of the following three:

(1) an alignment (correct or incorrect) of $x[n]$ vs. $y[m]$.

(2) an alignment of $x[n]$ vs. a blank

(3) an alignment of a blank vs. $y[m]$

Sub problem: optimal alignment of $x[1, \cdots, i]$ with $y[1, \cdots, j]$, for every combination of $0 \leqslant i \leqslant n$, $0 \leqslant j \leqslant m$.
$\Longrightarrow$ Fully characterized by i, j.

Define OPT(i,j) to be the minimum total alignment cost for aligning $x[1, \cdots, i]$ with $y[1, \cdots, j]$.

Recurrence relation for OPT:

```
OPT(i,j) = min(OPT(i-1,j-1) + B*[1 if x[i] != y[j], 0 otherwise], OPT(i-1,j)+A, OPT(i,j-1)+A) for i>=1, j>=1
[ OPT(0,0) = 0 [Base Case - subsumed by the others] ]
OPT(i,0) = i*A [extra base cases to avoid negative indices] OPT(0,j) = j*A [ "" ]
```

The optimum chooses the best of the three available options. For each of the three options, this describes the cost we worked out above.
If one of the strings is empty, the optimum (only solution) is to align the non-empty string against all blanks.

Tabular Implementation:

```
for (int i = 0; i <= n; i ++) a[i][0] = i*A;
for (int j = 0; j <= m; j ++) a[0][j] = j*A;
for (int i = 1; i <= n; i ++) {
     for (int j = 1; j <= m; j ++) {
          if (x[i] == y[j]) {
               a[i][j] = min(a[i-1][j-1], a[i-1][j] + A, a[i][j-1] + A);
          }
          else {
               a[i][j] = min(a[i-1][j-1] + B, a[i-1][j] + A, a[i][j-1] + A);
          }
     }
}
return a[n][m];
```

Final answer we need is OPT(n,m).

Correcness Proof: To prove: a[n][m] = OPT(n,m); [algorithm returns the minimum cost]
Statement we prove by induction: a[i][j] = OPT(i,j).
Induction on $i + j$, to ensure that whenever we need a value, it has a smaller "index" in the induction proof.
Base Case: $i + j = 0 \Longrightarrow i = j = 0 \Longrightarrow$ algorithm sets a[0][0] = 0 = OPT(0,0)
Induction Step: Assume that a[i'][j'] = OPT(i',j') whenever i'+j' < i+j [strong induction]

(1) **Case 1:** x[i] = y[j], and $i > 0$ and $j > 0$:

Algorithm writes

$$\begin{aligned} a[i][j] &= \min(a[i-1][j-1], a[i-1][j] + A, a[i][j-1] + A) \\ &= \min(OPT(i-1, j-1), OPT(i-1, j) + A, OPT(i, j-1) + A) \\ &\text{by IH, because } (i-1) + (j-1) < i + j, (i-1) + j < i + j, i + (j-1) < i + j \\ &= OPT(i, j) \qquad \text{by recurrence relation} \end{aligned}$$

(2) **Case 2:** x[i] != y[j], and $i > 0$ and $j > 0$

The algorithm writes

$$\begin{aligned} a[i][j] &= \min(a[i-1][j-1]+B, a[i-1][j]+A, a[i][j-1]+A) \\ &= \min(OPT(i-1,j-1)+B, OPT(i-1,j)+A, OPT(i,j-1)+A) \quad \text{By IH} \\ &= OPT(i,j) \quad \text{by recurrence relation} \end{aligned}$$

(3) **Case 3:** $i = 0$. The algorithm writes a[0][j] = A * j = OPT(0,j).

(4) **Case 4:** $j = 0$. The algorithm writes a[i][0] = A * i = OPT(i,0).

So we completed the induction step, and thus the proof. □

2 Knapsack Problem (Dynamic Programming)

Problem 2: Integer Knapsack Problem

Given n items. Item i has weight $w(i) > 0$ and value $v(i) \geqslant 0$. You have a wight limit of W on the total wight, and want to select a set S of wight at most W maximizing the total value from your set.

- **Fractional Knapsack**: can take a fraction of an item, use Greedy.
- **Integer Knapsack**: either take the item or not, Greedy not work.

Try Dynamic Programming. Define sub problem i to be items $\{1, \cdots, i\}$.
The optimum solution for items $1, \cdots, i$ either includes item i or does not include item i.
We need a definition of sub problems. To solve this, we have two parameters i and w. Sub problem is characterized by i items, and w, remaining weights we can choose from.

Definition 2.1: OPT (i, w)

Now we define OPT (i, w) : maximum value we can get from items $\{1, \cdots, i\}$ with total wight at most w.

Two cases:

(1) Case 1: i is excluded. Then OPT(i, w) = OPT$(i-1, w)$.

(2) Case 2: i is included. Then OPT(i, w) = $v(i)$ + OPT$(i-1, w-w(i))$.

Optimum doe the better of the two, so

```
if(w >= w(i)){
     OPT(i,w) = max(OPT(i-1,w), v(i) + OPT(i-1, w-w(i))); // Either choose i or not choose i if weight allow
}
else{
     OPT(i,w) = OPT(i-1,w); // We cannot choose i if weight does not allow
}
OPT(0,w) = 0; // Base Case
```

Now we have **Tabular Implementation**:

```
for (w = 0; w <= W; w++){
      a[0][w] = 0;
}
for (i = 1; i <= n; i++ ){
      for (w = 0; w <= W; w++){
            if (w >= w[i]){
                  a[i][w] = max(a[i-1][w], v[i] + a[i-1][w-w[i]]);
            }
            else {
                  a[i][w] = a[i-1][w];
            }
      }
}
return a[n][W];
```

We can also use a b[][] with only 0 and 1 to mark if i is added into the knapsack.

Correctness statement: a[n][W] = OPT(n,W)

Intermediate statement: a[i][w] = OPT(i,w) for all i, w.

Proof by induction on i. We don't have do $i + w$, because each time we look up values, they are for $i - 1$. Also, this means that weak induction is enough.

Induction step uses the assignment by the algorithm, then the induction hypothesis for a[i-1][w] and a[i-1][w-w[i]] and then the recurrence relation.

Note: If we only want the $a[][]$ values, not the $b[][]$ entries, we don't need the full $n \times W$ array. We only reference entries $i - 1$ for computing i. So it's enough to have a $2 \times W$ array. □

Runtime: $\Theta(nW)$. The running time of $O(nW)$ "looks" polynomial, but it is actually not.

Polynomial running time would mean polynomial in the input size.

But writing down the number W only takes $O(\log W)$ bits/characters. So the running time of W is $W = 2^{\log_2(W)}$, which is exponential in $\log_2(W)$. Thus, the running time is not polynomial.

We want a meaningful way to distinguish this type of exponential runtime from the 2^n runtime of exhaustive search. Intuitively, we want to say "If the numbers w_i and W are all 'small', then the algorithm is good/polynomial."

Definition 2.2: Pseudo-Polynomial Time

An algorithm A runs in pseudo-polynomial time if for every input, if all the numbers in the input were written in unary, the algorithm would run in polynomial time in the input size.

Example 2.3: Unary.

$$8 = 11111111$$

Notice this does not mean anyone would actually write numbers in unary. It is only a mathematically precise way of saying the algorithm runs fast when all the input numbers are reasonably small.

Note: We would not expect a genuinely polynomial-time algorithm for Knapsack, because - as we will see - Knapsack is NP-hard.