CPSC 500
October 21, 2013

Professor Condon on
Dynamic Programming
Scribe: Alireza Shafaei

Contents

1 Introduction

Optimization problems, they are eveywhere! If you haven't seen them before, then you were not paying attention. Remember those days that you had many deadlines ahead and you needed to efficiently schedule your tasks? In a simple case, you know how much time a task will take; but in a more complex case, you don't have enough time to do all of them perfectly, so you'd be interested in doing them partially, so that the expected value is maximized. Of course another strategy will be to procrastinate, which is still basically maximizing the duration of convenience. As you can see, depending on the value of each action (consciously or subconsciously), your solution may change. Study of optimization is beyond the scope of this week's material, but what we're about to discuss can help us solve a group of optimization problems.

2 Overview

In this lecture we will be discussing three problems known as *Minimum Edit Distance*, *Sequence Alignment*, and *Local Sequence Alignment*. Our approach to solving these problems will be *Dynamic Programming*.
We begin with solving the *Minimum Edit Distance* problem, and will propose a recursive function for it. This recursive function is later used to solve the problem using *Dynamic Programming* approach.

In the problem of *Sequence Alignment*, we will see how a series of decisions can be optimally made through *Dynamic Programming* given a scoring function.
In *Local Sequence Alignment*, we try to find the best subseries of decisions; which leads to a slightly varied method of solving the problem.

3 Minimum Edit Distance

We first begin by trying to find a solution to the following problem. Given two strings $A = a_1 \dots a_n$ and $B = b_1 \dots b_m$, find the minimum number of insert and delete operations necessary to transform A to B.

Example:

$$A = howdy, B = throw$$
$$A \xrightarrow{+t} thowdy \xrightarrow{+r} throwdy \xrightarrow{-y} throwd \xrightarrow{-d} throw = B$$

Observations. When we are trying to solve a problem, it's always good to analyze the answers and their attributes. This essentially leads us to developing algorithms that are capable of generating those answers. A typical observation in this problem, is the fact that, if $a_n = b_m$, we don't have to worry about them and we can focus on solving the problem for $A = a_1 \dots a_{n-1}$ and $B = b_1 \dots b_{n-1}$. This reduction in size of input, leaves us with the same problem that we had before, which hopefully could be easier to solve. Another important observation is that, if $a_n \neq b_m$, we only have two choices. We can either remove a_n from A, or we can add b_m to A; these two choices both reduce our problem size, but which one should be applied? Since we don't know which to choose, we have to make that decision at run time. We'd simply take the option which has least cost, using min function.
These observations can be summarized as follows:

- if $a_n = b_m$ then $solve(a_1 \dots a_{n-1}, b_1 \dots b_{m-1})$
- if $a_n \neq b_m$ optimal solution is either
  - delete $a_n \to solve(a_1 \dots a_{n-1}, b_1 \dots b_m) + 1$
  - add $b_m \to solve(a_1 \dots a_n, b_1 \dots b_{m-1}) + 1$

This neat definition already allows us to describe a dirty recursive function to solve the problem, but we don't want to do that! Through simple analysis of complexity, you can see that, this function will be taking exponential time to solve a problem. The main reason for that is the fact that you'd be solving same problems multiple times. So instead of going top-down, we'd rather solve this problem in a bottom-up fashion so that we only solve each subproblem once. This approach takes us to a widely used technique named *Dynamic Programming.*

Let $F(i, j)$ denote the minimum edit distance of $a_1 \dots a_i$ and $b_1 \dots b_j$.

Base Case. Just like the recursive methods, when we want to propose a dynamic programming solution to a problem, we should clearly specify the base cases. In case of recursive, the base case will be like the end of recursion; while in dynamic programming, we'd be building the next blocks on these base cases.
$F(0, 0)$, represents the problem of finding the minimum edit distance of two null strings. Luckily

it's obvious the answer is 0. $F(i, 0)$ and $F(0, j)$, denote the problem when one of the strings are null. The solution to these basic problems is obvious as well, because we either remove i, or add j characters. These observations can be formally written as:

$$F(0,0) = 0$$
$$F(i,0) = i \text{ when } 1 \leq i \leq n$$
$$F(0,j) = j \text{ when } 1 \leq j \leq m$$

General Case. Now for the general case, we simply have to put together the observations we had in the beginning.

$$F(i,j) = \begin{cases} F(i-1, j-1) & \text{if } a_i = b_j \\ min\{F(i-1.j), F(i, j-1)\} + 1 & \text{if } a_i \neq b_j \end{cases}$$

Now that we have put together the building blocks of the answer, we have to start the calculations. $F(i, j)$ could be simply visualized as a table. The initial state of this table will be something like the table 1.

		0	1	2	3	4	5	...
		-	x	x	x	x	x	...
0	-	0	1	2	3	4	5	...
1	x	1	?	?	?	?	?	...
2	x	2	?	?	?	?	?	...
3	x	3	?	?	?	?	?	...
4	x	4	?	?	?	?	?	...
5	x	5	?	?	?	?	?	...

Table 1: Initial state of $F(i, j)$ for minimum edit distance solution.

Now we need to figure the values marked with question mark. These values have to be calculated using the formula presented at *General Case*. Given the dependency of the presented formula on previous answers, the only value we can calculate at this stage is $(1, 1)$. After that, we can either proceed to calculating $F(1, 2)$ or $F(2, 1)$ (or even both in parallel!). After we've finished this table, the answer we're looking for will be at $F(n, m)$. Notably, this approach only takes $\theta(nm)$ time to solve the problem.

Example. In this example we have A =DARVAD and B =RAVAD. In table 2 you can see the final state of the program.

At this point, we know that the minimum edit distance for our problem will be 3, but you might be wondering how it can be produced? By a simple modification of the original program which solves this problem, we can leave a trace of the decisions we have made throughout the run. The decisions will be made either through the first criteria of general case, which is for when $a_n = b_m$ where we'd be using the value from the upper left side, or the case that we have removed or added a character, which will be either up side neighbour, or the left side. In table 2, values shown with boldface are the trace of our decision.

		0	1	2	3	4	5	6
		-	D	A	R	V	A	D
0	-	**0**	**1**	2	3	4	5	6
1	R	1	**2**	3	2	3	4	5
2	A	2	3	**2**	**3**	4	3	4
3	V	3	4	3	4	**3**	4	5
4	A	4	5	4	5	4	**3**	4
5	D	5	4	5	6	5	4	**3**

Table 2: Demonstration of $F(i,j)$ when A =DARVAD and B =RAVAD.

4 Sequence Alignment

Deoxyribonucleic acid (DNA) is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses [1].

In biological applications, there are times that we have extracted DNA sequence of two different living organisms, and we're interested in finding similarities; since they represent similar biological attributes. So the problem is to find the best alignment of these two sequences to maximize a matching criteria. This problem can be formally specified as follows. A DNA sequence is a string over alphabet set $\Sigma = \{A, C, G, T\}$. We have a cost table δ which describes the penalty and reward of aligning σ_1 with σ_2. Table 3, shows a sample δ table that we need to make decision throughout the formation of our solution.

As you can see, the value of aligning the same symbols is positive; and there's a penalty for

δ:	-	A	C	G	T
-	$-\infty$	-10	-10	-10	-10
A	-10	4	-6		
C	-10		5		
G	-10			5	
T	-10				4

Table 3: Sample matching criteria for application of DNA sequence alignment.

aligning different symbols. Interestingly, aligning different symbols may impose a different penalty. The goal is to align the sequence so that we maximize the value given by δ.

$$\begin{array}{ccccccc} - & - & T & - & C & G & A \\ A & C & T & A & G & G & A \end{array}$$

The score for this alignment is: $\delta(A,-)+\delta(C,-)+\delta(T,T)+\delta(A,-)+\delta(G,C)+\delta(G,G)+\delta(A,A)$. Let $F(i,j)$ be value of highest score alignment $X_1 \dots X_i$, and $Y_1 \dots Y_j$.

Base Case. For $F(0,0)$ the answer is obviously zero, and for $F(i,0)$ and $F(0,j)$, it will be the case that we are aligning all the symbols with $-$; so it could be written as follows.

$$F(0,0) = 0$$
$$F(i,0) = \sum_{k=1}^{i} \delta(X_k, -)$$
$$F(0,j) = \sum_{k=1}^{j} \delta(-, Y_k)$$

General Case. For the general case, we always have to make decision between three choices. We can align X_i, Y_j, and have the score of $\delta(X_i, Y_j) + F(i-1, j-1)$, or we can skip the symbols on X or Y, which will give the score of $\delta(X_i, -) + F(i-1, j)$ and $\delta(-, Y_j) + F(i, j-1)$.

$$F(i,j) = Max \begin{cases} \delta(X_i, Y_j) + F(i-1, j-1) & \text{when matching } X_i, Y_j \\ \delta(X_i, -) + F(i-1, j) & \text{when skipping } X_i \\ \delta(-, Y_j) + F(i, j-1) & \text{when skipping } Y_i \end{cases}$$

Having the base case and general case defined, we can proceed to calculations. If $|X| = n$ and $|Y| = m$, we'd be expecting the answer to be $F(n, m)$. To actually produce this value, we can again leave a trace of the decisions we've made throughout the calculations.

5 Local Sequence Alignment

The local sequence alignment is basically the same problem as sequence alignment, except that we don't want to align the whole sequences, instead we're trying to align a substring of them. In the original problem, the final score highly depended on the final alignment. But there are cases that we have two highly different sequences, which can be only partially aligned. In such case, the full alignment does not necessarily include the best partial alignment.
Let $F(i,j)$ be the optimal alignment of a suffix of $X_1 \dots X_i$ and a suffix of $Y_1 \dots Y_j$.

Base Case.

$$F(0,0) = 0$$
$$F(i,0) = 0$$
$$F(0,j) = 0$$

It's obvioust that in these cases, the best partial match will be having a score of 0.
General Case.

$$F(i,j) = Max \begin{cases} \delta(X_i, Y_j) + F(i-1, j-1) & \text{when matching } X_i, Y_j \\ \delta(X_i, -) + F(i-1, j) & \text{when skipping } X_i \\ \delta(-, Y_j) + F(i, j-1) & \text{when skipping } Y_i \\ 0 & \text{when we can't increase score} \end{cases}$$

The general case is similar to the original sequencing problem, except that we consider 0 as an option. When we start to align symbols, if we have a negative score, it means that we haven't matched anything yet; or if we did, its score is negative, so we might as well not consider that subsequence and try to align the rest of the sequence. When we add 0 to the options, we'll always have the freedom to forget about past alignments when they're not useful anymore ($\equiv$ getting negative score). The final answer will be maximum value in the set $F = \{F(i, j)\}$.

6 Conclusions

In order to learn how *Dynamic Programming* can be applied, we analyzed solutions to three different problems. Our analysis generally followed this pattern.

1. Observe the attributes of the solution.
2. Define a recursive function to find the solution.
3. Find the answer with bottom-up approach.
4. Trace the solution path to produce the actual sequence.

So when do we try to solve a problem with *Dynamic Programming*? In order to answer that, one has to go over the problems again and try to identify the common attributes. Usually, *Dynamic Programming* is known to work best, if the following conditions are satisfied for a problem.

1. The problem is to optimize.
2. A problem instance can be split into smaller pieces.
3. Given the answers to smaller instances, we can define the answer to bigger instances.
4. Subproblems of the problem overlap.

When the first three conditions are met, we say the problem has an **optimal substructure**. The fourth condition is to ensure that, the bottom-up approach is going to be efficient. The reason that a recursive function in this situation may run in exponential time, is because of the overlapping subproblems that grow exponentially. If the subprolems are not overlapping, it may be more efficient to solve the problem through recursion.

References

[1] Wikipedia. Dna. `http://en.wikipedia.org/wiki/DNA`, October 2013.