4
Dynamic Programming

Notes by Anupam Gupta

Please read the 15-451 lecture notes on dynamic programming for the basic concepts, of top-down dynamic programming (or memoization), and bottom-up dynamic programming. (It also talks about dynamic programming on trees, etc.) These notes here are focused on the issues of reducing space usage for these DPs.

4.1 Longest Common Subseqeuence

Here is the naive bottom-up dynamic program to find the longest common subseqeuence (LCS) of two strings S and T. Define M to be a table with $m+1$ rows and $n+1$ columns, where $M(i,j)$ computes the length of the longest common subsequence of the prefixes $S_{1:i}$ and $T_{1:j}$.

Algorithm 6: LCS-value(S,T)

$M(0,\star) = M(\star,0) = 0$
for $i = 1$ ***to*** m **do**
  for $j = 1$ ***to*** n **do**
    if $S_i = T_j$ **then**
      $M(i,j) \leftarrow 1 + M(i-1, j-1)$
    else
      $M(i,j) \leftarrow \max(M(i-1,j), M(i,j-1))$
return $M(m,n)$

Theorem 4.1. *Algorithm 6 computes the length of the longest common subsequence of two strings of length m,n in $O(mn)$ time and space.*

```
[0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 1 1 1 1 1 1 1 1]
[0 1 1 1 1 1 1 2 2 2 2]
[0 1 1 1 1 1 1 2 3 3 3]
[0 1 1 2 2 2 2 2 3 3 3]
[0 1 2 2 3 3 3 3 3 4 4]
[0 1 2 2 3 3 3 4 4 4 4]
[0 1 2 2 3 3 4 4 4 5 5]
[0 1 2 2 3 3 4 4 4 5 6]
```

Figure 4.1: The LCS of ACCTACAG and CATATACCAG.

4.1.1 Finding the LCS Itself

Having run Algorithm 6 to fill in the table, we can find the LCS itself in $O(m+n)$ time by just "following the decisions" when filling the table.

Algorithm 7: LCS-Search(S, T)

$i \leftarrow m, j \leftarrow n$
while $i > 0$ ***or*** $j > 0$ **do**
  if $S_i = T_j$ **then**
    output S_i
    $i \leftarrow i - 1, j \leftarrow j - 1$
  else
    if $M(i, j) = M(i-1, j)$ **then** $i \leftarrow i - 1$ **else** $j \leftarrow j - 1$

(Exercise: One of the strings T has been accidentally deleted, but you still have the string S, and the table $M(\cdot, \cdot)$. Show how to output the LCS in $O(m+n)$ time)

```
[0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 1 1 1 1 1 1 1 1]
[0 1 1 1 1 1 1 2 2 2 2]
[0 1 1 1 1 1 1 2 3 3 3]
[0 1 1 2 2 2 2 2 3 3 3]
[0 1 2 2 3 3 3 3 3 4 4]
[0 1 2 2 3 3 3 4 4 4 4]
[0 1 2 2 3 3 4 4 4 5 5]
[0 1 2 2 3 3 4 4 4 5 6]
```

Figure 4.2: The LCS of ACCTACAG and CATATACCAG is ATACAG.

4.2 *Space-Efficiency*

The above bottom-up algorithm for the LCS problem always takes $O(mn)$ time and space. A very recent result shows that the quadratic runtime is necessary in general, but we can reduce the space usage. The crucial observations are simple: (a) we care only about the value of $M(m, n)$, and (b) the update rule for a cell $M(i, j)$ depends only on $M(i-1, j-1)$, $M(i-1, j)$ and $M(i, j-1)$, which belong to the same row or previous row as the current cell (i, j) being filled in. Hence we can fill the table row-by-row, "keeping in mind" only rows $i-1$ and i when filling in row i. Formally, we define the table M to have only 2 rows and $n+1$ columns, and change the algorithm as follows:

Algorithm 8: Low-Space LCS(S, T)

$M(0, \star) = M(\star, 0) = 0$
for $i = 1$ ***to*** m **do**
  for $j = 1$ ***to*** n **do**
    if $S_i = T_j$ **then**
      $M(i \bmod 2, j) \leftarrow 1 + M(i-1 \bmod 2, j-1)$
    else
      $M(i \bmod 2, j) \leftarrow$
       $\max(M(i-1 \bmod 2, j), M(i \bmod 2, j-1))$
return $M(m \bmod 2, n)$

Theorem 4.2. *Algorithm 8 computes the length of the longest common subsequence of two strings of length m, n in $O(mn)$ time and $O(\min(m, n))$ space*

4.3 *(Optional) Finding the LCS in Linear Space*

How can we find the actual LCS using $O(m+n)$ space: clearly the search algorithm given in Algorithm 7 will no longer work, since we don't have the entire table. Hence we need to be smarter: the lovely idea here can be called "guess the mid-point".

This idea is essentially that used by Savitch for his classical result relating log-space computation to non-deterministic log-space.

The main observation is this: there exists a value q such that

$$LCS(S_{1:m}, T_{1:n}) = LCS(S_{1:m/2}, T_{1,q}) + LCS(S_{m/2+1,m}, T_{q+1,n}). \quad (4.1)$$

I visualize this as follows: when we follow the optimal solution up from $M(m,n)$ to $M(0,0)$, this optimal solution must cross row $m/2$ at some point—this point $(m/2, q)$ must provide this partition.

Now using Algorithm 8 on $S_{1:m/2}$ and T, and on the *reversed* strings $S_{m/2+1,m}$ and T, we can find the index q that achieves the equality (4.1). Now we can recurse on the two halves

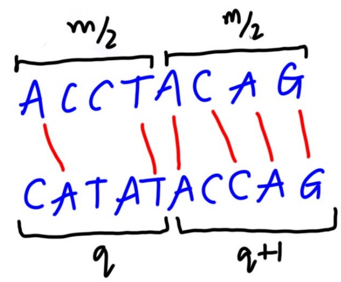

Figure 4.3: The choice of $q = 5$ gives the partition promised in (4.1).

Algorithm 9: Low-Space LCS-Search(S, T)

run Algorithm 8 on $S_{1:m/2}$ and T, and on reversed $S_{m/2+1,m}$ and T
find q that satisfies equality (4.1)
recurse on $S_{1:m/2}, T_{1:q}$, and on $S_{m/2+1,m}, T_{q+1,n}$.

Theorem 4.3. *Algorithm 9 runs in time $O(mn)$ and space $O(m+n)$.*

Proof. For the runtime, note that the first line of the algorithm runs in $O(mn)$ time, using Theorem 4.2. Now a linear-time scan can find the value q that minimizes the sum $LCS(S_{1:m/2}, T_{1,q}) + LCS(S_{m/2+1,m}, T_{q+1,n})$. Now for the inductive proof, assume that the rumtime of the recursive calls is at most $c(m/2)q + c(m/2)(n-q) = c(m/2)n$. Summing this all up, we get at most cmn. □

4.4 *Palindrome Deletion*

Starting with a string $S = s_0, s_1, \ldots, s_{n-1}$, you repeatedly delete contiguous substrings, each of which is a palinidrome. What is the fewest number of palindrome deletions which will turn S into the empty string?

Consider the string "`aacbbca`". It's possible to do it in three moves: remove the `aa` then `cbbc`, then `a`. However, if you remove `cbbc` first you are left with `aaa`, which can be removed in one move.

Here we given an $O(n^3)$ algorithm to do this. Define a recurrence variable $C[i,j]$ which is the minimial number of palindrome deletions required to obliterate the substring $s_i, \ldots, s_j$ *in isolation*. We'll derive a recurrence for this.

Consider a valid way of deleting the string by removing palindromes. For each character draw an arc (above the string) to the character it is paired with when it is deleted. These arcs are non-crossing. A character can be paired with itself.

So we consider a subrange form i to j. We'll compute the most efficient way to delete this using palindromes. We know that the character at i must be paired with some other character in $i \ldots j$. So we simply try all possibilities, and take the one that results in minimal cost. The one tricky thing is that if the arc is trivial (that is, it goes from i to i or from i to $i+1$) then it contributes a cost of 1. But if it is non-trivial then its cost is 0 because those two characters can be merged (as the first and last character) of the last palindrome that will be removed from the range $[i+1, j-1]$.

This leads to the following recurrence:

$$C[i,j] = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \displaystyle\min_{\substack{i \le k \le j \\ s_k = s_i}} \left\{ \begin{matrix} 1 & \text{if } k \in \{i, i+1\} \\ C[i+1, k-1] & \text{otherwise} \end{matrix} \right\} + C[k+1, j] & \text{otherwise} \end{cases}$$

There are $O(n^2)$ DP variables to compute, and each takes $O(n)$ time. So this algorithm runs in time $O(n^3)$ when memoized.

4.5 *Strokes*

You start with an array $A[]$ of length n consisting of all zeros. A sequence of *Stroke* operations is done on A. So Stroke(i, j, c) just does the assignments $A[k] \leftarrow c$ for all $i \le k \le j$. In other words it just sets that range of elements of $A[]$ to all be equal to c.

You're given $G = [g_0, \ldots, g_{n-1}]$, a list of positive numbers. Your goal is to use the minimum number of strokes to convert A to G, so at the end $A[i] = g_i$ for $0 \le i < n$.

We'll given an $O(n^3)$ algorithm to compute the fewest strokes required for a given input G.

Let $s[i,j]$ be the minimum number of strokes to color the range $[i,j]$ in total isolation. If $i > j$ then it's 0, if $i = j$ then it's 1. In the $i < j$ case we consider how we handle position i. One solution is to simply use a stroke to color i, but not use that stroke for anything to its right. This is $1 + s[i+1, j]$. On the other hand that same stroke can be used to color some others to the right of i. Let the first place that it also colors be at k. This option has stroke count:

$$\min_{\substack{i < k \le j \\ g_i = g_k}} 1 + s[i+1, k-1] + s[k, j] - 1.$$

The first 1 is for the stroke that colors i and k. The $s[i+1, k-1]$ is the cost of that range (which does not make use of the first stroke.). And $s[k, j] - 1$ is the cost of doing the $[k, j]$ range where you have a "free" stroke of color g_k coming from the left to color position k (and perhaps beyond) – thus the -1.

A useful way to visualize this is to think about drawing links above the pairs of elements whose final coloring is from the same stroke. These links cannot cross. Each group of connected links represents one stroke.

Here's the full recurrence for this solution:

$$s[i,j] = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ \min\{1 + s[i+1, j], \min\limits_{\substack{i < k \leq j \\ g_i = g_k}} s[i+1, k-1] + s[k, j]\} & \end{cases}$$

The running time to compute this recurrence is $O(n^3)$.

4.6 *Longest Path in a Tree*

This is an example of DP being applied to trees. Suppose you want to compute the longest path in an unrooted tree. We set up the DP algorithm by first rooting the tree, and computing a set c_i for each node i which are the children of that node.

What does a path in a rooted tree look like? It can start somewhere, go up (toward the root) for a while. Then turn around and go back toward a leaf. This leads to the following observation: Any path in a tree has a unique shallowest node.

A path whose shallowest node is i is said to *peak* at i. So we let L_i be the length of the longest path peaking at node i. And we let D_i be the length of the longest path going down from node i to a leaf.

Here are the recurrences:

$$D_i = \begin{cases} 0 & \text{if } c_i = \{\} \\ 1 + \max\limits_{j \in c_i} D_j & \text{otherwise} \end{cases}$$

$$L_i = \begin{cases} 0 & \text{if } c_i = \{\} \\ D_i & \text{if } |c_i| = 1 \\ D_{j_0} + D_{j_1} + 2 & \text{otherwise} \end{cases}$$

The term j_0 is the child of i with the biggest D value. And j_1 is the child of i with the second biggest D value. It's easy to see that these recurrences are correct, and can be computed in $O(n)$ time in a tree with n nodes.