**CS420+500: Advanced Algorithm Design and Analysis**

Lectures: Feb 27 + Mar 1 + Mar 3, 2017

*Prof. Will Evans* *Scribe: Adrian She*

In this lecture we:

- Summarized how linear programs can be used to model zero-sum games;
- Motivated/reviewed dynamic programming algorithms using the longest common subsequence problem;
- Developed a fast algorithm for longest increasing subsequence using a variant of dynamic programming.

# 1 Zero Sum Games: Summary

Consider a game with two players Row and Col with payoff matrix:

$$\begin{bmatrix} 3 & -1 \\ -2 & 1 \end{bmatrix}$$

Our goal is to find a mixed strategy (probability distribution on possible plays) for Row which maximizes their expected winnings no matter what strategy Col chooses to execute. Let $z$ denote the winnings of Row player using strategy $(x_1, x_2)$. Then the solution to the following linear program describes the optimal strategy for Row, since based on any strategy Row chooses, Col may choose a pure strategy which maximize Row's winnings:

$$\begin{aligned} &\max z \text{ subject to} \\ &z \leq 3x_1 - 2x_2 \text{ (Col chooses pure strategy 1)} \\ &z \leq -x_1 + x_2 \text{ (Col chooses pure strategy 2)} \\ &x_1 + x_2 = 1 \\ &x_1, x_2 \geq 0 \end{aligned} \tag{1}$$

On the other hand, Col wants to find a mixed strategy $(y_1, y_2)$ which minimizes their loss, no matter what pure strategy Row chooses. The solution to the following linear program describes the optimal strategy to minimize Col's losses:

$$\begin{aligned} &\min w \text{ subject to} \\ &w \geq 3y_1 - y_2 \text{ (Row chooses pure strategy 1)} \\ &w \geq -2y_1 + y_2 \text{ (Row chooses pure strategy 2)} \\ &y_1 + y_2 = 1 \\ &y_1, y_2 \geq 0 \end{aligned} \tag{2}$$

Note now that linear programs (1) and (2) are dual to each other!

| Row | Col |
|---|---|
| $\max 0x_1 + 0x_2 + 1x_3$ | $\min 0y_1 + 0y_2 + 1y_3$ |
| $-3x_1 + 2x_2 + x_3 \leq 0$ | $-3y_1 + y_2 + y_3 \geq 0$ |
| $x_1 - x_2 + x_3 \leq 0$ | $2y_1 - y_2 + y_3 \geq 0$ |
| $x_1 + x_2 = 1$ | $y_1 + y_2 = 1$ |
| $x_1, x_2 \geq 0$ | $y_1, y_2 \geq 0$ |

Hence, they have a common optimum $v$ by the linear programming duality theorem. The solutions are $(x_1, x_2) = (\frac{3}{7}, \frac{4}{7})$ and $(y_1, y_2) = (\frac{2}{7}, \frac{5}{7})$ respectively, which achieve optimum (expected winnings) $v = \frac{1}{7}$. This point is known as the **Nash equilibrium** for the game, as neither player can improve their expected winnings (or losses) as a result of changing their strategy at this point.

# 2 Dynamic Programming and Friends

We now switch gears to dynamic programming.

## 2.1 Longest Common Subsequence

**Problem** Given two character strings $X$ of $m$ characters, and string $Y$ of $n$ characters, what is a longest common subsequence of $X$ and $Y$? Denote a longest common subsequence of $X, Y$ by LCS(X,Y).

This is the longest non-crossing matching in a graph $B$ whose nodes are the characters of $X, Y$ respectively, and where an edge is drawn between a character of $X$ and a character of $Y$ if they are the same.

**Definition 1.** *A* **subsequence** *of a character string (array) $X$ is $X[i_1]X[i_2]...X[i_k]$ where $i_1 < i_2 < ... < i_k$*

For instance if X = ABANDON and Y = BADNODNO, then LCS(X,Y) is one of BADON, BANDO, or BANON.

We may be tempted to use a greedy strategy by directly aligning $X, Y$ and collecting all common characters between $X, Y$ as the longest common subsequence. However, this method does not work. Consider the strings $X = axxbxx$ and $Y = bxxaxxbxx$:

$$axxbxx$$
$$bxxaxxbxx$$

The longest subsequence found through direct alignment is $xxxx$, but the longest common subsequence is actually $axxbxx$. By considering strings $X = (axxbxx)^n$ and $Y = bxx(axxbxx)^n$, the length of the longest subsequence found using direct alignment may be much shorter than the actual longest common subsequence as $n$ increases.

Hence, we will attempt to use dynamic programming to solve this problem. The first step is to formulate recursive subproblems. To do this, we consider the relationship between X[m] and Y[n]:

- If X[m] = Y[n], we will match them and add it to LCS(X[1... m-1], Y[1...n-1]).
- If not, we consider the longer of LCS(X[1...m-1], Y[1..,n]) and LCS(X[1...m], Y[1...n-1]) as our LCS up to that point.

Hence, we may $mn$ subproblems of form LCS(X[1...i], Y[1...j]) for $1 \leq i \leq m$ and $1 \leq j \leq n$. To keep track of the solution of these subproblems, we use a table $F$, where the entry $F[i, j]$ denotes the length of LCS(X[1...i], Y[1...j]). We compute $F[i, j]$ by:

$$F[i,j] = \begin{cases} F[i-1, j-1] + 1 & X[i] = Y[j] \\ \max(F[i-1, j], F[i, j-1]) & X[i] \neq Y[j] \\ 0 & i = 0 \\ 0 & j = 0 \end{cases} \tag{3}$$

Note that $F[i, j]$ where $i = 0$ or $j = 0$ are the base cases. Using the above recurrence relation (3), the dynamic programming algorithm for longest common subsequence is then to initialize a $m \times n$ table $F$, compute each entry of $F$ using equation (3), then backtrack through the table to reconstruct the longest common subsequence.

| | ∅ | A | B | A | N | D | O | N |
|---|---|---|---|---|---|---|---|---|
| ∅ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 0 | 1 | 2 | 2 | 2 | 2 | 2 |
| D | 0 | 0 | 1 | 2 | 2 | 3 | 3 | 3 |
| N | 0 | 0 | 1 | 2 | 3 | 3 | 3 | 4 |
| O | 0 | 0 | 1 | 2 | 3 | 3 | 4 | 4 |
| D | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 4 |
| N | 0 | 0 | 1 | 2 | 3 | 4 | 4 | 5 |
| O | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 5 |

Table 1: Dynamic Programming Table for Example Strings

It was suggested that only two rows of the table need to be kept (instead of all rows) in order to save space. Furthermore, the algorithm takes $O(mn)$ time to compute the longest common subsequence of two strings.

Finally we note that longest common subsequence may be used the solve edit distance.

**Edit Distance** Given two strings $A$ of $n$ characters and $B$ of $m$ characters, what is the fewest number of insertions and deletions needed to transform $A$ into $B$?

This is because the edit distance between $A, B$ is $n + m - 2l$ where $l$ is the length of LCS(A,B).

## 2.2 Longest Increasing Subsequence

Now we consider the longest increasing subsequence problem.

**Longest Increasing Subsequence** Given a sequence of numbers $R$ of length $n$, find the longest subsequence in $R$ which is increasing.

For instance, if $R$ is the sequence 5,3,4,9,6,2,1,8, then the increasing subsequence would be 3,4,6,8.

There is a reduction from longest increasing subsequence to longest common subsequence. Sort $R$ and call the sorted list $R_{sorted}$. Then LCS(R, $R_{sorted}$) is the longest increasing subsequence in $R$. This takes $O(n^2)$ time.

We may provide a faster solution, motivated by "scanning" the array for all possible subsequence, which may be an initial attempt at solving this problem.

When finding the longest increasing subsequence of R[1...k], what do we want to keep about the array R[1...k-1]?

- We may want to keep the "best" or "most extensible" subsequence of R[1...k-1] as we scan the $k^{th}$ element of the array, namely the longest increasing sequence with smallest final element. However, consider the array R = [1 2 10 9 7 8 3 4 5]. The most extensible sequence up to the $6^{th}$ element would be S = [1 2 7 8], although we cannot extend S as we scan the rest of the array. At the same time, S is not the longest increasing subsequence since S' = [1 2 3 4 5] is longer.
- We will then keep the family of best increasing subsequences of lengths 1,2,3..,j. Call them BIS[1], ..., BIS[j] respectively.

For instance of R = [8 3 4 9 6 2 1 5 7 6], then the family of best increasing subsequences are as follows:

| After seeing | R[7]=1 | R[8] = 5 | R[9] = 7 |
|---|---|---|---|
| BIS[1] | 1 | 1 | 1 |
| BIS[2] | 3,4 | 3,4 | 3,4 |
| BIS[3] | 3,4,6 | 3,4,5 | 3,4,5 |
| BIS[4] | does not exist | does not exist | 3,4,5,7 |

Updating the family of best increasing subsequences if the whole subsequence is stored may take $O(n)$ time, however, we can do the update in logarithmic time. To do this, we consider BIS[i] as the list of end numbers of the best increasing subsequences of length $i$, as the numbers are encountered in a left to right sweep of the input. There is a pointer from an element $x$ in BIS[i] to $y$ in BIS[i-1] if $x$ is part of a sequence which extends $y$. For instance, the following would be the tree generated on the sample input R.

Storing the last elements instead of the full sequence enables adding the next number of R in $O(\log n)$ time. The last numbers of each BIS[i] are in sorted order, so we may perform binary search on i to determine whether to put the next number, and we point it to the end value of the previous BIS. This yields a $O(n \log n)$ time algorithm for longest increasing subsequence.

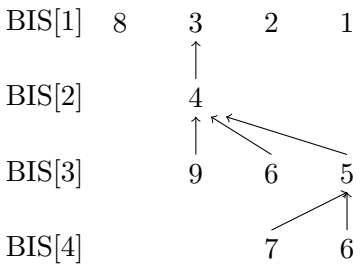


Figure 1: Collection of BIS[i] when algorithm finishes processing R

### 2.2.1 Reduction to LCS

Now, we can use longest increasing subsequence to create an algorithm for longest common subsequence.

We start with the two strings $X, Y$ and build a table, noting the positions in the strings whose characters match.

| | A | B | C | B | A | C | C | B |
|---|---|---|---|---|---|---|---|---|
| B | | x | | x | | | | x |
| C | | | x | | | x | x | |
| D | | | | | | | | |
| A | x | | | | x | | | |
| B | | x | | x | | | | x |
| C | | | x | | | x | x | |
| C | | | x | | | x | x | |

Table 2: Sample table for comparison to two strings

Finding a longest common subsequence of the two strings then means finding a sequence of index pairs of matching characters, for which the sequence increases in both coordinates.

We may write the matching index pairs of the example as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| r | 1 | 1 | 1 | 2 | 2 | 2 | ... |
| c | 2 | 4 | 8 | 3 | 6 | 7 | ... |

We are now faced with a problem, since running longest increasing subsequence on $c$ may return multiple matches per row (and we may not match a single character in $X$ with multiple characters in $Y$). Reverse the order of coordinates per row in c resolves this problem, as it then enforces picking at most one character in c per row in r.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| r | 1 | 1 | 1 | 2 | 2 | 2 | ... |
| c | 8 | 4 | 2 | 7 | 6 | 3 | ... |

Running longest increasing subsequence on the reversed sequence of column numbers then provides the longest common subsequence of $X, Y$. This takes $O(n^2 \log n)$ time, for there may be $O(n^2)$ matching pairs, although if the number of matching pairs is small, then this may be faster.

Recall that longest common subsequence can be used to solve the edit distance problem. The above approach of building a table of matching characters can be used to solve the edit distance problem as well (Myers 1986). We construct a graph from the table of matching characters by connecting $(i, j)$ with $(i, j + 1), (i - 1, j)$ with weight 1, since we need to make an insertion or deletion in this case. Next, we connect $(i, j), (i - 1, j - 1)$ with weight 0 iff the $i^{th}$ character of X and the $j^{th}$ character of Y match. These edge weights are based on our discussion of the recurrence relation for longest common subsequence.

Then the edit distance of $X, Y$ corresponds to the shortest path between $(0, 0)$ and $(n, m)$ where $n = |X|, m = |Y|$ in the LCS graph. We may run then run Dijkstra's algorithm to compute this path, which will have length $D$. At this point, Dijkstra's algorithm will have explored only those vertices $(i, j)$ that are at most distance $D$ from the origin. Since a horizontal or a vertical edge has length one, the coordinates must satisfy $|i - j| \leq D$. In an $n \times m$ array, there are $D \min\{n, m\}$ such vertices. Furthermore, Djikstra's algorithm runs in time $O(|V| + |E|)$ since the edge weights of the LCS graph are bounded. Hence, Myers' algorithm run time in time $O(\min\{n, m\}D)$ since the edge weights in the LCS graph are bounded each vertex is connected to at most two other vertices.