EECS3101 notes::Dynamic programming examples part 2

Jerry Wu

2023-03-07

More practice on dynamic programming

How to prove optimal substructure

Consider a subproblem A that is composed of subproblems $B_1, B_2, \ldots, B_n$ where $n \in \mathbb{N}$. **Optimal substruction** means that an optimal solution to problem A is expressed with **only optimal solutions** of $B_1, B_2, \ldots, B_n$.
The **cut and paste argument** entails the following::

- **Proof by contradiction::** Suppose that a solution to A can be expressed with a **sub optimal** solution to subproblem B.
- We can "cut out" said solution , then "paste in" the optimal solution to B in order to construct a solution to A that is **valid and more optimal** than the optimal sulution. Clearly this is a contradiction.

Matrix chain multiplication

Recall that if we multiply matrices $A_i \ldots A_k \ldots A_j$, we only need to know the optimal solutions from multiplying A_i A_k and A_k to A_j, where $k \in (i, j) \subset \mathbb{N}$.

$$OPT(i, j) = \neg OPT(i, k) + OPT(k, j)$$

The proof is trivial because we can simply remove the $\neg OPT(i, k)$ and replace it with $OPT(i, k)$ by way of common sense, because there is **always a more optimal solution** than something that is not optimal.

Two array maximum sum

"Good job catching my mistakes"-Larry YL Zhang 2023

Given two arrays of coins of varying positive values (i.e. $A[0 \dots n-1]$ and $B[0 \dots n-1]$), we select coins such that::

- No two coins are adjacent in the same array
- No two coins are from the same index
- The sum of the selected coins is maximized.

Example::

- $A = [9, 3, 2, 7, 3]$
- $B = [5, 8, 1, 4, 5]$

What would be the optimal solution in this case?

Solution

"This motivates us to find a more refined solution"-Larry YL Zhang 2023

Our first job is to identify the sub problems of our main problem. Let us take the first i elements of both arrays. We can start by finding $ans[i] = \max\{A[0 \dots i]\}$ and $\max\{B[0 \dots i]\}$. So can we construct $ans[i+1]$?
Define the array $DP_A[i] = \max\{A[i]\}$ to be the set of solutions for all elements from 0 to i, and $DP_B[i] = \max\{B[i]\}$. We also need to consider the case where neither $A[i]$ or $B[i]$ are chosen as candidates (i.e. $DP_N[i] = \max\{x \notin A \wedge x \notin B\}$). So we can begin construction of our recurrence relations for our three cases::

- For $A[i]$ (max sum up to index i with $A[i]$ selected)::

$$DP_A[i] = \max\{DP_B[i-1] + A[i], DP_N[i-1] + A[i]\}$$

- For $B[i]$ (max sum up to index i with $B[i]$ selected)::

$$DP_B[i] = \max\{DP_A[i-1] + B[i], DP_N[i-1] + B[i]\}$$

- For $N[i]$ (max sum up to index i with neither $A[i]$ or $B[i]$ selected)::

$$DP_N[i] = \max\{(DP_A[i-1], DP_B[i-1])\}$$

We have 3 recurrence relations, each with $O(n)$ time complexity, and we can spend $O(1)$ space to find the max of each number. So our algorithm is $O(n)$ overall, which is very efficient.

Exercise:: The array DP_N is not necessary. Can we get rid of it? *The proof is trivial and is left as an exercise to the reader.*

> *"Believe it or not, I'm actually trying to confuse you less"-Larry YL Zhang 2023*

Longest common subsequence (LCS)

Given two strings X and Y, find the length of the longest common subsequence.

- Z is a subsequence of X ($Z \subset X$), if it is possible to generate Z by skipping 0 or more characters in X.
- For example: What is $LCS(X, Y)$?

$$X = "ACGGTTA00", Y = "CGTAT00"$$

Solution

Let us start by identifying our subproblems. We can take substrings of both X and Y::

$$X_\Sigma = substring(X, i, j), Y_\Sigma = substring(Y, k, m) | i, j, k, m \in \mathbb{N}$$

We have 4 parameters, so we'll need a 4D array (which is not ideal).

> *"We are 3 dimensional creatures. We don't like 4 dimensional."-Larry YL Zhang 2023*

So how can we make parameter(s) into non parameters? Yes. We can fix the starting points of both arrays at 0. So now we have that::

$$X_\Sigma = substring(X, 0, i), Y_\Sigma = substring(Y, 0, j), i, j \in \mathbb{N}$$

So our problem as a whole can be defined as::

$$ans[i][j] = LCS(X_\Sigma, Y_\Sigma)$$

We now have a few cases to consider, namely the following::

- $ans[i-1][j-1]$
- $ans[i-1][j]$
- $ans[i][j-1]$

We can represent this decision structure by some pseudocode

```
def main():
    ans[i][j]=length of LCS of X[0...i] and Y[0...j]
    if X[i]==Y[j]:
    //case 1
        return ans[i-1][j-1] + 1
    if X[i] != Y[j]:
    //case 2
        return max(ans[i-1][j],ans[i][j-1])
```

So our space taken is $O(|X| \cdot |Y|)$.

The knapsack problem

Given n items, each having an integer weight w_i and value v_i where $i \in [1, n]$ and given a knapsack with weight capacity W. Select items to fill the knapsack such that the total value is maximized.

There are two versions of this problem::

- Continuous:: can take any real valued fraction of any item. A greedy algorithm would work.
- Discrete:: Each item is either taken or not taken. Would require dynamic programming.

We will revisit this problem in the next lecture.