Lecture 20: Dynamic Programming II

Lecture Overview

• 5 easy steps
• Maximum value contiguous subsequence
• Text justification
• Parent pointers

Summary

* DP ≈ “controlled brute force”
* DP ≈ guessing + recursion + memoization
* DP ≈ dividing into reasonable # subproblems whose solutions relate — acyclicly — usually via guessing parts of solution.
* time = # subproblems \times \text{time per subproblem}
  treating recursive calls as \(O(1)\)
  (usually mainly guessing)

  • essentially an amortization
  • count each subproblem only once; after first time, costs \(O(1)\) via memoization
* DP ≈ shortest paths in some DAG

5 Easy Steps to Dynamic Programming

1. define subproblems \(\text{count # subproblems}\)
2. guess (part of solution) \(\text{count # choices}\)
3. relate subproblem solutions \(\text{compute time/subproblem}\)
4. recurse + memoize \(\text{time = time/subproblem} \cdot \text{# subproblems}\)
   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: = a subproblem
   OR by combining subproblem solutions        ⇒ extra time

Examples:                     Fibonacci                                  Shortest Paths
subprobs:                      \( F_k \)                                      \( \delta_k(s,v) \) for \( v \in V \), \( 0 \leq k < |V| \)
                              for \( 1 \leq k \leq n \)    = \({\text{min path using} \leq k \text{ edges}}\)
# subprobs:               \( n \)                                                       \( V^2 \)
guess:                          nothing                                       edge into \( v \) (if any)
# choices:                    1                                                        indegree(\( v \)) + 1
recurrence:                   \( F_k = F_{k-1} \)                          \( \delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v) \mid (u,v) \in E\} \)
                             + \( F_{k-2} \)                                    time/subpr:
                              \( \Theta(1) \)                                                        \( \Theta(1 + \text{indegree}(v)) \)
topo. order:                  for \( k = 1, \ldots, n \)                        for \( k = 0, 1, \ldots |V| - 1 \) for \( v \in V \)
                              \( \Theta(n) \)                                                        \( \Theta(VE) \)
orig. prob.:                   \( F_n \)                                                        \( \delta_{|V|-1}(s,v) \) for \( v \in V \)
extra time:                   \( \Theta(1) \)                                                        \( \Theta(V) \)

Maximum Value Contiguous Subsequence

We wish to find a contiguous subsequence of numbers in an array \( A[1..n] \) that sum up to the maximum value possible. This problem is only interesting if the values of the elements can be negative – else we can simply choose all the non-negative numbers in the array.

Find \( i \) and \( j \) where \( i < j \) to maximize:

\[
\sum_{l=i}^{j} A[l]
\]

Clearly we can solve this in \( O(n^2) \) time by sweeping values for \( i \) and \( j \) keeping \( i < j \).

Can we do better? Yes, by using DP!

\[
M(j) : \text{max sum over all windows ending with } j
\]

\[
M(j) = \max\{M(j - 1) + A[j], A[j]\}
\]
\[
M(0) = 0
\]
The first term corresponds to extending the window ending at position \( j - 1 \). The second corresponds to starting a new window containing only \( A[j] \).
We are not quite done after computing all the \( M(j) \)'s. The value of the solution is not necessarily \( M(n) \), but rather

\[
\text{optimal value} = \max_{l=1}^{n} M(l)
\]

We have \( n \) subproblems each of which take constant time to compute, for an overall complexity of \( \Theta(n) \).

**Text Justification**

Split text into “good” lines

- obvious (MS Word/Open Office) algorithm: put as many words that fit on first line, repeat

- but this can make very bad lines

![Figure 1: Good vs. Bad Text Justification.](image)

- Define \( \text{badness}(i, j) \) for line of words \([i : j]\).
  For example, \( \infty \) if total length > page width, else \((\text{page width} - \text{total length})^3\).

- goal: split words into lines to min \( \sum \text{badness} \)

1. subproblem = min. badness for suffix words \([i : j]\)  
   \( \Rightarrow \) # subproblems = \( \Theta(n) \) where \( n = \# \text{ words} \)

2. guessing = where to end first line, say \( i : j \)  
   \( \Rightarrow \) # choices = \( n - i = O(n) \)

3. recurrence:
• \(\text{DP}[i] = \min(\text{badness}(i, j) + \text{DP}[j] \text{ for } j \text{ in range } (i + 1, n + 1))\)

• \(\text{DP}[n] = 0\)
  \[\implies\] time per subproblem = \(\Theta(n)\)

4. order: for \(i = n, n - 1, \ldots, 1, 0\)
  total time = \(\Theta(n^2)\)

\[
\begin{align*}
&i \rightarrow \bigcirc \quad \bigcirc \quad \bigcirc \quad j \\
\text{badness}(i,j)
\end{align*}
\]

Figure 2: DAG.

5. solution = \(\text{DP}[0]\)

**Parent Pointers**

To recover actual solution in addition to cost, store **parent pointers** (which guess used at each subproblem) & walk back

• typically: remember argmin/argmax in addition to min/max

• example: text justification

\[
\begin{align*}
(3)' \quad \text{DP}[i] &= \min(\text{badness}(i, j) + \text{DP}[i][0], j) \\
&\text{ for } j \text{ in range}(i+1, n+1) \\
\text{DP}[n] &= (0, \text{None}) \\
(5)' \quad i &= 0 \\
&\text{while } i \text{ is not None:} \\
&\quad \text{start line before word } i \\
&\quad i = \text{DP}[i][1]
\end{align*}
\]

• just like memoization & bottom-up, this transformation is **automatic**
  no thinking required