Dynamic Programming III

Lecture Overview
- Text Justification
- Parenthesization
- Knapsack
- Pseudopolynomial Time
- Tetris Training

Readings
CLRS 15

Review:
* DP is all about subproblems & guessing
* 5 easy steps:
  (a) define subproblems: count \( \notin \) subprobs.
  (b) guess (part of solution): count \( \notin \) choices
  (c) relate subprob. solutions: compute time/subprob.
  (d) recurse + memoize OR build DP table bottom up:
      time = time/subprob. \( \times \notin \) subprobs
      (check subproblems related acyclically)
  (e) check original problem = a subproblem or solvable from DP table ( \( \Longrightarrow \) extra time)
* for sequences, good subproblems are often prefixes OR suffixes OR substrings
Dynamic Programming III

Text Justification:

Split text into “good lines”

- obvious (MS Word/Open Office) algorithm: put as many words fit on first line, repeat
- but this can make very bad lines

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

- define badness$(i, j)$ for line of words $[i : j]$ e.g.,
  \[
  \begin{cases}
  \text{if total length > page width} \\
  \text{(page width - total length)}^3 \\
  \end{cases}
  \]

- goal: split words into lines to min $\sum$ badness

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

2. guessing = where to end first line, say $i : j$
   \[\Rightarrow \text{choices} = n - i = O(n)\]

3. relation:
   - $DP[i] = \min(\text{badness}(i, j) + DP[j] \text{ for } j \text{ in range}(i + 1, n + 1))$
   - $DP[n] = \phi$
     \[\Rightarrow \text{time per subproblem} = O(n)\]

4. total time = $O(n^2)$

5. solution = $DP[\phi]$
   (& use parent pointers to recover split)
Parenthesization:

Optimal evaluation of associative expression - e.g., multiplying rectangular matrices

\[\begin{array}{c}
A \quad B \quad C
\end{array}\]

(AB)C costs \(\Theta(n^2)\)
A(BC) costs \(\Theta(n)\)

Figure 2: Evaluation of an Expression

2. guessing = outermost multiplication \(\frac{\cdots}{\text{1}_{k-1}}(\cdots)\)

\[\text{\# choices } = O(n)\]

1. subproblems = prefix & suffix? NO

\[\text{= cost of substring } A[i : j]\]

\[\Rightarrow \text{\# subproblems } = \Theta(n^2)\]

3. Relation:

- \(DP[i, j] = \min(DP[i, k] + DP[k, j] + \text{cost of multiplying } (A[i] \cdots A[k - 1]) \text{ by } (A[k] \cdots A[j - 1])) \text{ for } k \text{ in range}(i + 1, j)\)

- \(DP[i, i + 1] = \phi\)

\[\Rightarrow \text{cost per subproblem } = O(n)\]

4. total time = \(O(n^3)\)

5. solution = \(DP[0, n]\)

(& use parent pointers to recover parent)

Knapsack:

Knapsack of size \(S\) you want to pack

- item \(i\) has integer size \(s_i\) & real value \(v_i\)

- goal: choose subset of items of maximum total value subject to total size \(\leq S\)

First Attempt:

1. subproblem = value for suffix \(i\): \text{WRONG}

2. guessing = whether to include item \(i\) \(\Rightarrow\) \# choices = 2

3. relation:
Dynamic Programming III

- \(DP[i] = \max(DP[i+1], v_i + DP[i+1] \text{ if } s_i \leq S^*)\)
- not enough information to know whether item \(i\) fits - how much space is left? GUESS!

1. subproblem = value for suffix \(i\):
   given knapsack of size \(X\)
   \(\Rightarrow\) \# subproblems = \(O(nS)\) !

3. relation:
   - \(DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X - s_i] \text{ if } s_i \leq X)\)
   - \(DP[n, X] = \phi\)
     \(\Rightarrow\) time per subproblem = \(O(1)\)

4. total time = \(O(nS)\)

5. solution = \(DP[\phi, S]\)
   (& use parent pointers to recover subset)
   **AMAZING:** effectively trying all possible subsets!

Knapsack is in fact NP-complete! \(\Rightarrow\) suspect no polynomial-time algorithm (polynomial in length of input).

What gives?
- here input = \(< S, s_0, \cdots, s_{n-1}, v_0, \cdots, v_{n-1} >\)
- length in binary: \(O(\lg S + \lg s_0 + \cdots) \approx O(n \lg \ldots)\)
- so \(O(nS)\) is not "polynomial-time"
- \(O(nS)\) still pretty good if \(S\) is small
- "pseudopolynomial time": polynomial in length of input & integers in the input

Remember:
- polynomial - GOOD
- exponential - BAD
- pseudopoly - SO SO
Dynamic Programming III

![Figure 3: Tetris](image)

**Tetris Training:**
- given sequence of $n$ Tetris pieces & a board of small width $w$
- must choose orientation & $x$ coordinate for each
- then must drop piece till it hits something
- full rows do not clear
  without these artificialities WE DON'T KNOW! (but: if $w$ large then NP-complete)
- goal: survive i.e., stay within height $h$

[material below covered in recitation]

**First Attempt:**

1. subproblem — survive in suffix $i$: WRONG
2. guessing = how to drop piece $i$ $\Rightarrow$ $\sharp$ choices = $O(w)$
   What do we need to know about prefix : $i$?

1. subproblem = survive? in suffix $i$:
   given initial column occupancies $h_0, h_1, \cdots, h_{w-1}$
   $\Rightarrow$ $\sharp$ subproblems = $O(n \cdot h^w)$
3. relation: $DP[i, h] = \max(DP[i, m]$ for valid moves $m$ of piece $i$ in $h)$
   $\Rightarrow$ time per subproblem = $O(w)$

4. total time = $O(nwh^w)$
5. solution = $DP[\phi, \phi]$
   (& use parent pointers to recover moves)