CS 312: Algorithms

Intro to Dynamic Programming

Dan Sheldon

Mount Holyoke College

Last Compiled: October 31, 2018

Algorithm Design Techniques

- Greedy
- Divide and Conquer
- Dynamic Programming
- Network Flows

Learning Goals

	Greedy	Divide and Conquer	Dynamic Programming
Formulate problem			
Design algorithm		✓	✓
Prove correctness	✓		
Analyze running time		✓	
Specific algorithms	✓		✓

Weighted Interval Scheduling

- Television scheduling problem: Given n shows with start time s_i and finish time f_i, watch as many shows as possible, with no overlap.
- A Twist: Each show has a value v_i and want a set of shows S, with no overlap and maximum value $\sum_{i \in S} v_i$.
- Greedy? Example on board.
- Problem formulation
  - Show (job) j has value v_j, start time s_j, finish time f_j
  - Assume shows sorted by finishing time $f_1 \leq f_2 \leq \ldots \leq f_n$
  - Shows i and j are compatible if they don't overlap
  - **Goal**: selected subset of non-overlapping jobs with maximum value

Dynamic Programming Recipe

- Step 1: Devise simple recursive algorithm
  - Make *one decision* by trying all possibilities
  - Use a recursive solver to evaluate the value of each
  - Problem: it does redundant work, often exponential time
- Step 2: Write recurrence for optimal value
- Step 3: Design iterative algorithm

Step 1: Recursive Algorithm

- **Observation**: Let O be the optimal solution. Either $n \in O$ or $n \notin O$. In either case, we can reduce the problem to a *smaller instance* of the same problem.
- Recursive algorithm to find value of optimal subset of first j shows

Compute-Value(j)

  Base case: if $j = 0$ return 0

  Case 1: $j \in O$
  Let p_j be highest-numbered show compatible with j
  val1 $= v_j +$ Compute-Value(p_j)

  Case 2: $j \notin O$
  val2 $=$ Compute-Value($j - 1$)

  return max(val1, val2)

Running Time?

- Board work
- **Problem**: running time is exponential in n (recursion tree). But redundant work is done. Only n unique subproblems.

Step 2: Recurrence

- Recurrence = shorter, mathematical, description of recursive structure for optimal value
- Let OPT(j) be the value of optimal subset of first j jobs
- Let p_j be highest-numbered job that is compatible with j

$$\mathrm{OPT}(0) = 0$$
$$\mathrm{OPT}(j) = \max\{\underbrace{v_j + \mathrm{OPT}(p_j)}_{\text{Case 1}}, \quad \underbrace{\mathrm{OPT}(j-1)}_{\text{Case 2}}\}$$

Step 3: Iterative "Bottom-Up" Algorithm

WeigthedIS
  Initialize array $M[0 \dots n]$ to hold optimal values
  $M[0] = 0$ ▷ Value of empty set
  for $j = 1$ to n **do**
    $M[j] = \max(v_j + M[p_j], M[j-1])$
  end for

- Example execution
- Running time? $O(n)$
- Usually direct "wrapping" of recurrence in appropriate for loop. Pay attention to dependence on previously-computed entries of M to know which direction to iterate.

Review

- Recursive algorithm → recurrence → iterative algorithm

Epilogue: Recovering the Solution (1)

Idea: modify the algorithm to what choice is made at each iteration

WeigthedIS
  Initialize array $M[0 \dots n]$ to hold optimal values
  Initialize array choose$[1 \dots n]$ to hold choices
  $M[0] = 0$
  for $j = 1$ to n **do**
    $M[j] = \max(v_j + M[p_j], M[j-1])$
    Set choose$[j] = 1$ if first value is bigger, and 0 otherwise
  end for

Epilogue: Recovering the Solution (2)

Then trace back from end and "execute" the choices

  Use algorithm above to fill in M and choose arrays
  $O = \{\}$
  $j = n$
  while $j > 0$ **do**
    if choose(j) == 1 **then**
      $O = O \cup \{j\}$
      $j = p_j$
    else
      $j = j - 1$
    end if
  end while

Dynamic Programming Recipe

- Step 1: Devise simple recursive algorithm
  - Make *one decision* by trying all possibilities
  - Use a recursive solver to evaluate the value of each
  - Problem: it does redundant work, often exponential time
- Step 2: Write recurrence for optimal value
- Step 3: Design iterative algorithm

Dynamic Programming Outlook

- First example: Weighted Interval Scheduling
  - Binary first choice: $j \in O$ or $j \notin O$?
- Next time: rod-cutting
  - First choice has n options