Dynamic Programming Examples

Sam McCauley

April 5, 2024

Welcome Back!

- Assignment 4 in last night
- Assignment 5 out tonight
  - Group assignment
  - Last assignment before midterm—make sure you get good practice
  - Last question might rely on material from Monday depending on how far we get
- Should mostly finish with DP today; start Network Flows Monday
- Questions?

Knapsack

Today: Weight limit only

Knapsack

- You are packing a bag, with a weight capacity C
- You have a collection of items to put in your bag
- Each item i has a weight w_i and a value v_i (both nonnegative integers)
- Choose a subset of items with *total weight* at most C
- Goal: maximize the *total value* of the items you pack

Knapsack

From Last Class:

- Does greedy work? How could we greedily pack a bag?
- Option 1: pick the highest-value item. Counterexample?
- Option 2: pick the lowest-weight item. Counterexample?
- Option 3: pick the item maximizing value/weight. Counterexample?

Recursive Knapsack

- Goal for the next portion of class: come up with the dynamic program for knapsack together [On Board #1]
- There are likely to be some false starts! I'm not writing the solution line by line.
- (Also there are some ideas that don't work that I specifically want to discuss :) so we may circle back to some suggestions)

Recursive Knapsack Solution

- Subproblem: (i, c): what is the largest-value solution among the first i items with total weight at most c?
- Memoization structure: $n \times (C+1)$ matrix (storing $OPT(i, c)$ for $i \in \{1, \ldots, n\}$ and $c \in \{0, \ldots, C\}$.
- Recurrence: $OPT(i, c) = \max\{OPT(i-1, c), v_i + OPT(i-1, c-w_i)\}$ if $w_i \leq c$
  $OPT(i, c) = OPT(i-1, c)$ otherwise.
- Final answer: $OPT(n, C)$
- Before moving forward: what subproblems do we need to solve in order to fill in $OPT(i, c)$?
  - In what order should we fill out the table?
  - Base cases?
  - Answer: we need all entries in $OPT(i-1, c)$ to fill out any entry in $OPT(i, c)$. So go item by item. Our base case must fill out all entries in $OPT(1, c)$.

Recursive Knapsack Solution

- (recall) Memoization structure: $n \times (C+1)$ matrix (storing $OPT(i,c)$ for $i \in \{1,\ldots,n\}$ and $c \in \{0,\ldots,C\}$).
- Evaluation order: Row-major order (row by row: fill in $OPT(i,c)$ for $c \in \{0,\ldots,C\}$ before filling in $OPT(i+1,c)$ for $c \in \{1,\ldots,C\}$).
- Base cases: $OPT(1,c) = v_1$ if $c \geq w_1$, $OPT(1,c) = 0$ if $c < w_1$.
- Space: $O(nC)$ Time: $O(nC)$

A Comment on Running Time

- Running time is $O(nC)$
- In algorithms we generally want a "polynomial" running time (i.e. a polynomial in the *size* of the input). All running times we've seen so far in this class were polynomial.
- Is this polynomial in the size of the input?
  - No! The size of the input is $O(n + \log_2 C)$ (it takes $\log_2 C$ bits to write C down)
  - C is exponential in $\log_2 C$. So this running time is not polynomial
- This knapsack DP is pseudopolynomial: the running time is polynomial in the *value* of the input, not the *size*

Pseudopolynomial Running Time Comments

- When is pseudopolynomial running time a big downside?
- Is this a practical problem?
- What happens when the weights of the items are not integers? Does our DP work? Can we make it work?

Knapsack: Recent Developments

- STOC 2024 papers announced last week (top theory conference)
- *FIVE* papers on Knapsack:
  - Two that parameterize by the size of the largest item (if largest item has size w can get $O(n + w^2 \log^4 w)$ time
  - Two that give an arbitrarily good *approximations* for knapsack
  - One approximation algorithm for "partition": the special case where $C = \frac{1}{2} \sum w_i$