Recitation 15

Dynamic Programming

Dynamic Programming generalizes Divide and Conquer type recurrences when subproblem dependencies form a directed acyclic graph instead of a tree. Dynamic Programming often applies to optimization problems, where you are maximizing or minimizing a single scalar value, or counting problems, where you have to count all possibilities. To solve a problem using dynamic programming, we follow the following steps as part of a recursive problem solving framework.

How to Solve a Problem Recursively (SRT BOT)

1. **Subproblem** definition subproblem $x \in X$
   - Describe the meaning of a subproblem **in words**, in terms of parameters
   - Often subsets of input: prefixes, suffixes, contiguous subsequences
   - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** subproblem solutions recursively $x(i) = f(x(j), \ldots)$ for one or more $j < i$
3. **Topological order** to argue relation is acyclic and subproblems form a DAG
4. **Base** cases
   - State solutions for all (reachable) independent subproblems where relation doesn't apply/work
5. **Original problem**
   - Show how to compute solution to original problem from solutions to subproblems
   - Possibly use parent pointers to recover actual solution, not just objective function
6. **Time** analysis
   - $\sum_{x \in X} \text{work}(x)$, or if $\text{work}(x) = O(W)$ for all $x \in X$, then $|X| \cdot O(W)$
   - $\text{work}(x)$ measures **nonrecursive** work in relation; treat recursions as taking $O(1)$ time

Implementation

Once subproblems are chosen and a DAG of dependencies is found, there are two primary methods for solving the problem, which are functionally equivalent but are implemented differently.

- A **top down** approach evaluates the recursion starting from roots (vertices incident to no incoming edges). At the end of each recursive call the calculated solution to a subproblem is recorded into a memo, while at the start of each recursive call, the memo is checked to see if that subproblem has already been solved.
- A **bottom up** approach calculates each subproblem according to a topological sort order of the DAG of subproblem dependencies, also recording each subproblem solution in a memo so it can be used to solve later subproblems. Usually subproblems are constructed so that a topological sort order is obvious, especially when subproblems only depend on subproblems having smaller parameters, so performing a DFS to find this ordering is usually unnecessary.

Top down is a recursive view, while Bottom up unrolls the recursion. Both implementations are valid and often used. Memoization is used in both implementations to remember computation from previous subproblems. While it is typical to memoize all evaluated subproblems, it is often possible to remember (memoize) fewer subproblems, especially when subproblems occur in 'rounds'.

Often we don't just want the value that is optimized, but we would also like to return a path of subproblems that resulted in the optimized value. To reconstruct the answer, we need to maintain auxiliary information in addition to the value we are optimizing. Along with the value we are optimizing, we can maintain parent pointers to the subproblem or subproblems upon which a solution to the current subproblem depends. This is analogous to maintaining parent pointers in shortest path problems.

Exercise: Simplified Blackjack

We define a simplified version of the game **blackjack** between one **player** and a **dealer**. A **deck of cards** is an ordered sequence of n cards $D = (c_1, \ldots, c_n)$, where each card c_i is an integer between 1 and 10 inclusive (unlike in real blackjack, aces will always have value 1). Blackjack is played in **rounds**. In one round, the dealer will draw the top two cards from the deck (initially c_1 and c_2), then the player will draw the next two cards (initially c_3 and c_4), and then the player may either choose to draw or not draw one additional card (a hit).

The player wins the round if the **value** of the player's hand (i.e., the sum of cards drawn by the player in the round) is ≤ 21 and exceeds the value of the dealer's hand; otherwise, the player loses the round. The game ends when a round ends with fewer than 5 cards remaining in the deck. Given a deck of n cards with a **known order**, describe an $O(n)$-time algorithm to determine the maximum number of rounds the player can win by playing simplified blackjack with the deck.

Solution:

1. **Subproblems**
   - Choose suffixes
   - $x(i)$: maximum rounds player can win by playing blackjack using cards $(c_i, \ldots, c_n)$
2. **Relate**
   - Guess whether the player hits or not
   - Dealer's hand always has value $c_i + c_{i+1}$
   - Player's hand will have value either:
     - $c_{i+2} + c_{i+3}$ (no hit, 4 cards used in round), or
     - $c_{i+2} + c_{i+3} + c_{i+4}$ (hit, 5 cards used in round)
   - Let $w(d, p)$ be the round result given hand values d and p (dealer and player)
     - player win: $w(d, p) = 1$ if $d < p \leq 21$
     - player loss: $w(d, p) = 0$ otherwise (if $p \leq d$ or $21 < p$)
   - $x(i) = \max\{w(c_i+c_{i+1}, c_{i+2}+c_{i+3})+x(i+4), w(c_i+c_{i+1}, c_{i+2}+c_{i+3}+c_{i+4})+x(i+5)\}$
   - (for $n - (i-1) \geq 5$, i.e., $i \leq n-4$)
3. **Topo**
   - Subproblems $x(i)$ only depend on strictly larger i, so acyclic
4. **Base**
   - $x(n-3) = x(n-2) = x(n-1) = x(n) = x(n+1) = 0$
   - (not enough cards for another round)
5. **Original**
   - Solve $x(i)$ for $i \in \{1, \ldots, n+1\}$, via recursive top down or iterative bottom up
   - $x(1)$: the maximum rounds player can win by playing blackjack with the full deck
6. **Time**
   - # subproblems: $n + 1$
   - work per subproblem: $O(1)$
   - $O(n)$ running time

Exercise: Text Justification

Text Justification is the problem of fitting a sequence of n space separated words into a column of lines with constant width s, to minimize the amount of white-space between words. Each word can be represented by its width $w_i < s$. A good way to minimize white space in a line is to minimize **badness** of a line. Assuming a line contains words from w_i to w_j, the badness of the line is defined as $b(i, j) = (s - (w_i + \ldots + w_j))^3$ if $s > (w_i + \ldots + w_j)$, and $b(i, j) = \infty$ otherwise. A good text justification would then partition words into lines to minimize the sum total of badness over all lines containing words. The cubic power heavily penalizes large white space in a line. Microsoft Word uses a greedy algorithm to justify text that puts as many words into a line as it can before moving to the next line. This algorithm can lead to some really bad lines. LaTeX on the other hand formats text to minimize this measure of white space using a dynamic program. Describe an $O(n^2)$ algorithm to fit n words into a column of width s that minimizes the sum of badness over all lines.

Solution:

1. **Subproblems**
   - Choose suffixes as subproblems
   - $x(i)$: minimum badness sum of formatting the words from w_i to w_{n-1}
2. **Relate**
   - The first line must break at some word, so try all possibilities
   - $x(i) = \min\{b(i, j) + x(j + 1) \mid i \leq j < n\}$
3. **Topo**
   - Subproblems $x(i)$ only depend on strictly larger i, so acyclic
4. **Base**
   - $x(n) = 0$ badness of justifying zero words is zero
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(0)$
   - Store parent pointers to reconstruct line breaks
6. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(n^2)$

- $O(n^3)$ running time
- Can we do even better?

Optimization

- Computing badness $b(i, j)$ could take linear time!
- If we could pre-compute and remember each $b(i, j)$ in $O(1)$ time, then:
- work per subproblem: $O(n)$
- $O(n^2)$ running time

Pre-compute all $b(i, j)$ in $O(n^2)$, also using dynamic programming!

1. **Subproblems**
   - $\boxed{x(i, j)\text{: sum of word lengths } w_i \text{ to } w_j}$
2. **Relate**
   - $x(i, j) = \sum_k w_k$ takes $O(j - i)$ time to compute, slow!
   - $x(i, j) = x(i, j - 1) + w_j$ takes $O(1)$ time to compute, faster!
3. **Topo**
   - Subproblems $x(i, j)$ only depend on strictly smaller $j - i$, so acyclic
4. **Base**
   - $x(i, i) = w_i$ for all $0 \leq i < n$, just one word
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up
   - Compute each $b(i, j) = (s - x(i, j))^3$ in $O(1)$ time
6. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

6.006 Introduction to Algorithms

Spring 2020