CSE541 Class 4

Jeremy Buhler

January 28, 2019

Today: even more dynamic programming (also, linear algebra)

1 A Mathematical Conundrum

- Who here has used MATLAB?
- It's a program for doing math with *matrices*!
- (Recall that an $m \times n$ matrix A is a 2D array of numbers a_{ij}, $1 \leq i \leq m$, $1 \leq j \leq n$)

- One of the things you can do with matrices is multiply them.
- Given matrices $A_{m \times p}$ and $B_{p \times n}$, product $C = A \cdot B$ is defined by

$$C_{ij} = \sum_{k=1}^{p} a_{ik} \times b_{kj}.$$

- (Recall that matrices must be conformable to multiply: # cols in first equals # rows in second)
- Total # scalar multiplies to compute C is mpn.
- We can similarly define multiplication for an arbitrary product of matrices $A_1 \cdot A_2 \cdot \ldots \cdot A_N$ of various sizes, provided each successive pair is conformable.

That's nice, but where's the problem?

- Suppose you ask Matlab to compute a matrix product ABC.
- Matrix multiplication is associative, so the program could compute ABC as $A \cdot (B \cdot C)$ or $(A \cdot B) \cdot C$.
- However, the total work may *not* be equal for different associative orderings!

- **Example**: $A_{2\times 12}$, $B_{12\times 3}$, $C_{3\times 4}$
- For $(AB)C$, we pay $2 \cdot 12 \cdot 3 + 2 \cdot 3 \cdot 4 = 96$ scalar multiplies.
- For $A(BC)$, we pay $12 \cdot 3 \cdot 4 + 2 \cdot 12 \cdot 4 = 240$ multiplies.
- **Problem**: given a chain of matrices $A_1 \dots A_N$ to multiply, where A_i has size $p_i \times q_i$, find an associative ordering that minimizes the total number of scalar multiplies.

2 A DP Approach

Why do DP?

- A greedy approach might choose least costly pair, most costly pair, leftmost pair, etc to do first.
- Unfortunately, none of these simple choices are optimal for all problem instances.
- Again, there is a small number of first choices that are clearly "complete" for optimality.

We will derive an ordering from outside in (pick last multiply first).

- Last multiply looks like

  $$(A_1 \cdot \ldots \cdot A_k)(A_{k+1} \cdot \ldots \cdot A_N)$$

  for some $1 \le k < N$.
- Each possible split point k is a choice; call it c_k.
- Some optimal solution must make choice c_k, for at least one k between 1 and N.
- Hence, our algorithm idea is: "compute best soln given *each* first choice c_k, then take best overall"

Let's work through the proof.

- **Complete Choice**: We consider all possible ways of splitting the chain into two parts, so one of them must be consistent with optimality.
- **Inductive Structure**: for each possible initial choice, we are left with *two* smaller subproblems equivalent to first problem, with no external constraints.
- (This is an extension vs. our previous work: multiple subproblems!)
- **Pf**: for each choice c_k, we are left with problems of multiplying $A_1 \dots A_k$ and $A_{k+1} \dots A_N$ with least cost.
- These problems are the same as the top-level problem, and any solution is compatible with the initial division into $(A_1 \dots A_k) \times (A_{k+1} \dots A_N)$.

- **Optimal Substructure**: Let $\Pi'_{k,j}$ be an optimal solution to the jth subproblem induced by initial choice c_k. Then combining these optimal sub-solutions with c_k yields a solution Π_k that is optimal *among all solutions that make choice* c_k.
- (Remember, we have to complete the proof for *each* c_k!)
- **Pf**: Let $\Pi'_{k,\ell}$ and $\Pi'_{k,r}$ be optimal associative orderings for the left and right subproblems.
- Cost of final multiply is $p_1 q_k q_N$ (computed from sizes of two subparts)
- Total cost of solution Π_k is therefore

$$\text{cost}(\Pi_k) = \text{cost}(\Pi'_{k,\ell}) + \text{cost}(\Pi'_{k,r}) + p_1 q_k q_N.$$

- This cost is separable, so we could stop here. But for reference, here is the standard contradiction argument.
- Suppose Π_k is not optimal among solns that split first at k.
- Let Π^*_k be a better solution that splits first at k, with subsolutions $\Pi^*_{k,\ell}$, $\Pi^*_{k,r}$.
- We have

$$\Pi^*_{k,\ell} + \Pi^*_{k,r} > \Pi'_{k,\ell} + \Pi'_{k,r}.$$

- Conclude that at least one of the subsolutions for Π^*_k is better than the corresponding subsolution for Π_k. $\rightarrow\leftarrow$
- Hence, Π_k is optimal given choice k. QED
- (Note that standard contradiction argument can still work on separable costs with a term for *each* subproblem!)

3 Writing the Recurrence

Let's go on to formalize the recursive algorithm as a recurrence.

- Let the vector $[i, j]$ denote the subproblem of finding the optimal parenthesization of $A_i \dots A_j$, $1 \le i \le j \le N$.
- Let $T[i, j]$ be the cost of an optimal solution to subproblem $[i, j]$.
- As we established through our optimal substructure property, if we make first choice c_k (i.e. divide product after A_k), we have that

$$T[i, j \mid k] = T[i, k] + T[k+1, j] + p_i q_k q_j.$$

- Moreover as we established through our complete choice property, $T[i, j] = \min_{i \le k < j} T[i, j \mid k]$.
- *Base case*: $T[i, i] = 0$ for any i, since no work is needed to compute a product of one matrix.
- *Goal*:If our full problem is to multiply $A_1 \dots A_N$, we want to compute $T[1, N]$.

4 Dynamic Programming Solution

OK, how should we order our subproblems to get a fast solution?

- Observe that $T[i,j]$ depends only on computation of Ts for strictly shorter subproblems.
- More precisely, we can compute $T[i,j]$ given $T[k,\ell]$ for $i \leq k < \ell \leq j$.
- These smaller subproblems $[k,\ell]$ are the dependencies of $[i,j]$.
- Suppose we order the subproblems by increasing length follows:

  for $d = 1 \ldots N-1$
      for $i = 1 \ldots N-d$
          compute $T[i, i+d]$ from its dependencies and store it

- When we compute $T[i,j]$, we have already computed all its dependencies, so this is a feasible subproblem ordering.

OK, let's do our final complexity analysis.

- Assume that we can look up stored $T[i,j]$ value in $O(1)$ time.
- Computing one subproblem minimizes over up to N constant-sized expressions on stored terms, so requires worst-case time $\Theta(N)$.
- Number of subproblems in the domain is $\Theta(N^2)$, since for each of the N values of i, there are up to N values of $j > i$.
- Overall cost of solving each subproblem once to obtain $T[1,N]$ is therefore $O(N^3)$!

One further comment: if you actually have to do matrix chain multiplication in practice, there is an $O(n \log n)$ algorithm to find the optimal order, due to Hu and Shing (1981). It is rather hairy.

5 Back to the Sack

Now, let's return to our old friend, the 0-1 knapsack problem.

- We are given n items.
- Item x_i has weight w_i, value v_i.
- We are also given a capacity W.
- We seek highest-value subset of items with total weight $\leq W$.
- Must take all or none of an item.

Recall that we failed miserably to apply greedy approach to this problem. Let's try dynamic programming!

- Put the items in *any* order you want.
- We will consider each item in this order.
- *Idea*: do we put *last* item x_n in knapsack? (Assumes it is feasible; if not, the answer is always no).
- **Complete Choice**: clearly, opt soln either does or does not contain item x_n, so one of two choices is consistent with optimality!
- **Inductive Structure**: let (S, W) be an instance of the knapsack problem, where S is the item set and W is the knapsack capacity.
- If we skip x_n, subproblem is $(S - \{x_n\}, W)$
- If we add x_n, subproblem is $(S - \{x_n\}, W - w_n)$
- Both subproblems can be solved arbitrarily while producing a solution compatible with our first choice.
- **Optimal Substructure**: suppose we solve the subproblem (S', W') optimally after making the first choice.
- If we skip x_n, value of solution is that of subproblem.
- If we add x_n, value of solution is that of subproblem plus v_n.
- In each case, apply standard contradiction argument.

OK, on to recurrence...

- What is our domain?
- Given the sorted order on S, general subproblem is $(\{x_1 \dots x_i\}, w)$ for a prefix $x_1 \dots x_i$ of S and $w \leq W$.
- Let's call this general subproblem $[i, w]$.
- (*Note*: provided item weights are rational, they can be considered integer, as can w, so $0 \leq w \leq W$ is a valid index set.)
- Let $V[i, w]$ be value of an opt solution for this subproblem.
- By substructure, we have that

$$V[i, w] = \begin{cases} \max(V[i-1, w], V[i-1, w - w_i] + v_i) & \text{if } w_i \leq w \\ V[i-1, w] & \text{otherwise} \end{cases}$$

- Base cases are $V[i, 0] = 0$ (since we cannot add to a full knapsack) and $V[0, w] = 0$ (since we have not yet added any items)
- Goal point is $V[n, W]$.
- To go bottom-up, consider 2D matrix of $0..n$ by $0..W$.

- Initially, fill every point's storage except bases with value $-\infty$.
- Compute recurrence in column-major order from top to bottom.

- (Note that unreachable points contribute $-\infty$ to the max and so are not used.)
- Domain size is $\Theta(nW)$, and each subproblem takes $\Theta(1)$ time.
- Hence, total running time is $\Theta(nW)$.

Is $\Theta(nW)$ a good result for worst-case efficiency? More next time...