Intro to Complexity Theory

1 More Uses of Dynamic Programming

1.1 Matrix Multiplication

Dynamic programming can drastically speed up the multiplication of $k \geq 3$ matrices. We want to compute $A_1 A_2 \ldots A_k$, where $A_i \in \mathbb{R}^{m_i \times n_i}$ $\forall i$. Order matters during multiplication! For example, take $A_1 \in \mathbb{R}^{1 \times n}, A_2 \in \mathbb{R}^{n \times 1}, A_3 \in \mathbb{R}^{1 \times n}$. If we multiply as $(A_1 A_2) A_3$, it will take $2n$ total computations, but if we multiply as $A_1(A_2 A_3)$, it will take a whopping $2n^2$ instead.

Our goal is to compute the optimal evaluation order for a sequence of matrices $A_i A_{i+1} \ldots A_j$. We know that there is some final multiplication $(A_i \ldots A_k)(A_{k+1} \ldots A_j)$ where each of the terms is likewise computed optimally. We say $OPT(i,j)$ is the optimum cost to compute the sequence, and for each computation, we solve the terms optimally and spend time to compute the resultant matrix.

$$OPT(i,j) = \min_{i \leq k \leq j-1} (OPT(i,k) + OPT(k+1,j) + n_i m_k m_j \text{ with } OPT(i,i) = 0$$

This can be implemented in a bottom-up nested loop, where the first loop runs to $j - i$ and the second to i. In other words, we compute the optimum way to compute the product of one matrix, then two, then three, and so on, eventually combining our previous optima for k matrices.

1.2 Sequence Alignment

Sequence alignment deals with strings that may have been changed via insertion, omission, or replacement. This shows up in a variety of applications, from spellchecking to DNA analysis. To compute sequence alignment, we inset spaces into s or t until the lengths are equal, then write them on top of each other. Then, the alignment cost is the sum of α for each mismatched letter and β for each letter matched with a space. We can also define the edit distance of strings s and t as the minimum cost of any sequence of operations transforming s into t, where α is the cost for replacing a character and β is the cost for inserting or removing a character.

Given an n-character string s and an m-character string t, compute the optimal alignment cost from s to t. We say $OPT(i,j)$ is the minimum alignment cost of $s[1 \ldots i]$ and $t[1 \ldots j]$. It must be the minimum of three terms:

$$\beta + OPT(i-1,j)\text{: matched last letter in } s \text{ against a space}$$

$$\beta + OPT(i,j-1)\text{: matched last letter in } t \text{ against a space}$$

$$OPT(i-1,j-1) + \alpha * (s[i] \neq t[j])\text{: matched } s \text{ and } t \text{ on top of each other and paid } \alpha \text{ if they didn't match up}$$

With base cases:

$$OPT(i,0) = i\beta \ \forall i$$

$$OPT(0,j) = j\beta \ \forall j$$

2 Complexity Theory

2.1 Reductions

Say that we have encountered some problem in research which we're pretty sure cannot be solved by a polynomial-time algorithm. **NP-completeness** deals with relating such a problem to known difficult problems, by using methodologies for showing that some (new) problem is at least as hard as lots of old problems that nobody knows how to solve efficiently. The key tool here is the **reduction**, also called the transformation, from an old problem X to a new problem Y. The reduction shows that there exists an algorithm that will solve X if it could also solve Y. In other words, it transforms Y into X in polynomial time. We write $X \leq pY$ if there exists a reduction from X to Y.

Assuming $X \leq pY$:

- If Y is solvable in polynomial time, so is X.
- If X cannot be solved in polynomial time, neither can Y.

2.2 Formalization

We define a **problem** X as a set of strings.

- Strings $s \in X$ called "Yes" instances
- Strings $s \notin X$ called "No" instances

For example, the MST problem is the set of all strings encoding a graph $\mathcal{G}$ with edge costs c_e and target cost C such that $\mathcal{G}$ has a spanning tree of cost $\leq C$.

An **algorithm** A **solves** X when $A(x) =$ "Yes" $\iff x \in X$. In other words, the algorithm only answers "Yes" if the answer was in fact "Yes", and the algorithm only answers "No" if the answer was in fact "No".

The strategy of decomposing optimization problems like "Find the MST" to decision problems like "Does an MST exist with cost $\leq C$" is beneficial for formalization. If we can solve the optimization problem, we can use it to answer the decision problem, so optimization is at least as hard as decision. Furthermore, if we can solve the decision problem, we can binary search to find the optimum cost, and in many (but not all) cases we can create different instances that allow us to actually solve the optimization problem. In many cases, optimization and decision are equivalent, but decision is easier to formalize and usually easier to solve.