LINMA2111 - CM5 - 2015

Mathieu Dath & Damien Scieur

25 avril 2015

In the last course, we were exploring dynamic programming : we had seen how to compute $\binom{n}{k}$. Let us now see how we can improve chained matrix multiplication.

1 Chained matrix multiplication

As stated in the last course, we have n matrices $M_1, ..., M_n$ and the dimension of the i-th matrix M_i is $d_{i-1} \times d_i$. We saw that the naïve cost of the product AB (with A being an $a \times b$ matrix and B an $b \times c$ matrix) is abc.

We showed that the number of possible orders is Catalan's number (approximately $\frac{4^n}{n\sqrt{n}}$) and brute force is thus clearly not an option.

1.1 Divide and conquer approach

Let us define $\text{Cost}[i,j]$ as the best cost to multiply $M_i, ..., M_j$. We divide with respect to the top bracketing $(M_i, ..., M_k)(M_{k+1}, ..., M_j)$; then we can further define $\text{Cost}[i,j]$ as

$$\text{Cost}[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_k \text{Cost}[i,k] + \text{Cost}[k,j] + d_{i-1}d_k d_j & \text{otherwise} \end{cases} \tag{1}$$

By this definition, every leaf of the recursion tree of $\text{Cost}[1,n]$ correspond to one order of multiplication of $M_1, ..., M_n$. This means that the number of leaves is equal to Catalan's number and thus that this approach has a time complexity of $\Omega(\frac{4^n}{n\sqrt{n}})$... This approach gives the same results as the brute force.

This is wasteful because intermediate results are recomputed over and over again in the recursion tree : the subproblems are overlapping.

1.2 Dynamical programming approach

Within this new approach, $\text{Cost}[i,j]$ is a table, where the entry $C[i,j]$ is the optimal number of operations for multiplying matrices $i, i+1, ...j-1, j$. Indeed, $i \leq j$.

This table is filled diagonal by diagonal (starting from the main diagonal then going up) using Equation (**??**). The time to compute Cost$[i, j]$ is $\Theta(j - i)$ and the time to fill in the table is thus

$$\begin{aligned}
\text{Fill in time} &= \sum_{i=1}^{n} \sum_{j=i}^{n} \Theta(j - i) \\
&= \sum_{i=1}^{n} \sum_{k=0}^{n-i} \Theta(k) = \sum_{i=1}^{n} \Theta((ni)^2) \\
&= \sum_{k}^{n-1} \Theta(k^2) = \Theta(n^3) \leq \Omega(4^n)
\end{aligned}$$

This is clealry better than the complexity we had before !

Note that fill in the table is a "bottom-up" approach while a recursive algorithm combined with a memory function (some memorization) is a "top down approach.

2 When can dynamical programming be used for optimisation problem ?

There are two principles to take into account :

Optimality principle For any optimal structure (like a list, a tree, etc.) answering the problem, the substructures (sublists, etc.) are optimal for a subproblem (a problem with smaller instances).

Subproblems are overlapping Divide and conquer is wasteful because it computes many times the same subproblem.

2.1 Exemples

Shortest path in a graph

We can see that a subpath of a shortest path is also a shortest path (thus the optimality principle is respected). Dijkstra's and Floyd's algorithms are good examples of dynamical programming applied to this problem, altough they are a bit more clever (see next chapter).

Knapsack

The problem here is to fill in a bag of volume V with objets among n, each having a volume v_i and a value c_i. The problem can be written

$$\max \sum_{i \in \text{Bag}} c_i$$
$$\text{s.t.} \sum_{i \in \text{Bag}} N_i \leq V$$

The brute force approach has a time complexity of $\mathcal{O}(2^n)$. But what about the dynamical programming approach ?

Let us define $\text{Value}(S, V)$ as the maximal value for the set of object S and a total bag volume V, then we can write

$$\text{Value}(S, V) = \max \big(\underbrace{\text{Value}(S \setminus \{i\}, V - v_i) + c_i}_{\text{take the object } i} ; \underbrace{\text{Value}(S \setminus \{i\}, V)}_{\text{do not take object } i} \big)$$

with i an arbitrary object of S.

Since we are interessed in :
– A solution with at most a weight V in the knapsack,
– which takes care of all n objects,
we need to compute $\text{Value}(n, V)$. But before computing $\text{Value}(n, V)$ we need to compute $\text{Value}(i, j)$ for all $i \leq n$ and $j \leq V$. Also, we need $\mathcal{O}(1)$ operation to compute one entry. We can thus deduce that the complexity of the algorithm is $\mathcal{O}(nV)$.

As some entries may never be needed to compute Value(Solution), it is a recursive function with memorization (a top-down approach may be useful).

Longuest common subsequence

Let us show the concept with an example : we start with the sequences

$$X = a\ b\ c\ b\ d\ a\ b$$
$$Y = b\ d\ c\ a\ d\ a$$

They have many common subsequences, but the longuest is $Z = bcda$:

$$X = a\ \underline{b\ c}\ b\ \underline{d\ a}\ b$$
$$Y = \underline{b}\ d\ \underline{c}\ a\ \underline{d\ a}$$

Let us take the following prefix :

$$X_i = X_1 ... X_i \text{ of } X_n = X_1 ... X_n$$
$$Y_i = Y_1 ... Y_i \text{ of } Y_n = Y_1 ... Y_n$$

and define $\text{LCS}(i,j)$ as the length of the longuest common subsequence of X_i and Y_j. We can write

$$\text{LCS}(0,0) = 0$$
$$\text{LCS}(i,j) = \begin{cases} \max\big(\text{LCS}(i-1,j);\text{LCS}(i,j-1)\big) & \text{if } x_i \neq y_j \\ \text{LCS}(i-1,j-1)+1 & \text{if } x_i = y_j \end{cases}$$

for $i, j > 0$.

The cost for X_n, Y_n is $\Theta(n^2)$, but for all these problems, we do not want only the optimal cost, but also the optimal structure (path, subsequence, etc.). We thus need to create a second table where we will record the optimal choices for each subproblem as it is then easy to reconstruct the optimal structure in the end.

3 Greedy algorithm

Let us now introduce a new kind of algorithm, which will be seen in more details in the next course : the greedy algorithms. In this section, we look at the activity selection problem : given a set of activities $\{a_1, ..., a_n\}$ and the knowledge that activity a_i takes the time interval $[s_i, f_i[$ (where s_i is the start time and f_i the end time), we want to find the maximum size set of mutually disjoint activities.

This is equivalent to booking an auditorium with as many activites as possible without scheduling conflicts.