Analysis of Algorithms

Homework 2 – Dynamic Programming

Abraham Murciano

December 28, 2020

1 Matrix Multiplication

We are given the following four matrices to be multiplied, as well as their sizes.

Matrix	Size
$\mathbf{A}_1$	10×30
$\mathbf{A}_2$	30×5
$\mathbf{A}_3$	5×60
$\mathbf{A}_4$	60×10

We are to apply the algorithm described in class in order to find the order in which to apply the associative matrix multiplications such that the number of scalar multiplications is minimal.

When multiplying two matrices, suppose these are of sizes $r \times s$ and $s \times t$, the resulting matrix would be of size $r \times t$, meaning $r \cdot t$ dot products must be calculated. And each of those dot products would be a sum of s scalar multiplications. Therefore the total number of scalar multiplications for these two matrices is $r \cdot s \cdot t$.

When multiplying a chain of n matrices, we can say that whatever the optimal order of performing the multiplications, if the last two matrices to be multiplied are

$$(\mathbf{A}_1 \times \cdots \times \mathbf{A}_k) \times (\mathbf{A}_{k+1} \times \cdots \times \mathbf{A}_n)$$

then the way to multiply $\mathbf{A}_1 \times \cdots \times \mathbf{A}_k$ must be optimal, and the same can be said about $\mathbf{A}_{k+1} \times \cdots \times \mathbf{A}_n$.

We denote by $m_{i,j}$ the minimal number of scalar multiplications required to multiply matrices $\mathbf{A}_i$ through $\mathbf{A}_j$, where the dimensions of matrix $\mathbf{A}_i$ is denoted by $d_i \times d_{i+1}$. Now we can define $m_{i,j}$.

$$m_{i,j} = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} (m_{i,k} + m_{k+1,j} + d_i \cdot d_{k+1} \cdot d_{j+1}) & i < j \end{cases}$$

Let $s_{i,j}$ be the value of k which gives the minimal number of multiplications in the definition of $m_{i,j}$. Now we are ready to compute the initial question by calculating $m_{1,4}$, $s_{1,4}$, and all the intermediate results, as shown in table 1.

i \ j	1	2	3	4
1	0	1500	4500	5000
2	-	0	9000	4500
3	-	-	0	3000
4	-	-	-	0

i \ j	1	2	3	4
1	-	1	2	2
2	-	-	2	2
3	-	-	-	3
4	-	-	-	-

Table 1: The computed final and intermediate results of $m_{1,4}$ (left) and $s_{1,4}$ (right)

Thus we have shown that the optimal way to multiply $\mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3 \times \mathbf{A}_4$ is $((\mathbf{A}_1)(\mathbf{A}_2))((\mathbf{A}_3)(\mathbf{A}_4))$ with a total of 5000 scalar multiplications.

2 Longest Non-Decreasing Subsequence

The problem. Given a sequence of natural numbers $S = (s_1, s_2, \ldots, s_n)$, we are to create an algorithm to find the longest monotonically non-decreasing subsequence as well as its length.

Some notation. Let us denote by X'_z the sequence of the first z elements of a sequence X. Additionally, let $X +\!\!+ Y$ be the concatenation of two sequences X and Y.

The optimal substructure. Suppose T is a solution; i.e. it is a non-decreasing subsequence of S of maximal length. Now suppose the last value of T'_m is s_a.

We can be certain that T'_m is the longest non-decreasing subsequence of S'_a (i.e. the first a terms of S) such that the last element of the subsequence is less than or equal to t_{m+1}.

The proof. Suppose T'_m is not the longest subsequence within the conditions specified earlier. Meaning there exists U which is a longer subsequence of S'_a. Then we would be able to substitute the prefix T'_m for the U in the subsequence T, and that would give a longer subsequence of S, contradicting the maximality of T.

The formula. Let T denote the non-decreasing subsequence of S of maximal length. Additionally, let Q_i denote the maximal subsequence of S'_i which ends

in s_i.

$$T = \max_{1 \le i \le n} (Q_i)$$

$$Q_i = \begin{cases} () & i = 1 \\ \max\limits_{j < i \ : \ s_j \le s_i} (Q_j) +\!\!+ (s_i) & i > 1 \end{cases}$$

In simpler terms, our solution T is obtained by first calculating a maximal subsequence which ends in each term s_i of S, then taking the longest of those.

And in order to find a maximal subsequence which ends in s_i, we first find a maximal subsequence within the terms before s_i, which ends in a value smaller than or equal to s_i, so that s_i may be concatenated to it.

The algorithm. The iterative algorithm LNDS calculates the longest non-decreasing subsequence of the sequence S. In this algorithm, p_i is set to the index of the predecessor of s_i in the sequence Q_i. (If the predecessor is set to null, that indicates that there is no predecessor.) Further, we let l_i denote the length of Q_i. Throughout the algorithm, each index i is inserted into a min-heap H_{l_i} whose key is s_i. We use min-heaps so we can calculate $\min(H_k)$ in constant time. The result, T, is a stack whose top element is the beginning of the subsequence and whose last is the end of the subsequence.

---

function LNDS(S)
  if $|S| = 0$ **then return** ()
  $p_1 :=$ null ▷ The first element has no predecessor
  $l_1 := 1$ ▷ The length of Q_1 is 1
  $H_1 \leftarrow (s_1, 1)$ ▷ Insert 1 into the min-heap of subsequences of length 1
  $h := 1$ ▷ h is the length of the longest Q_i
  for i from 2 to n **do**
    for k from h to 1 **do**
      $j := \min(H_k)$ ▷ j is the last index of a Q_j s.t. $j < i$ and $s_j < s_i$
      if $s_j \le s_i$ **then** break ▷ We found the correct j
    $p_i := j$
    $l_i := l_j + 1$
    $H_{l_i} \leftarrow (s_i, i)$
    $h := \max(h, l_i)$ ▷ Update largest length
  $j := \min(H_h)$ ▷ The end of the longest subsequence
  while $j \neq$ null **do**
    $T \leftarrow s_j$
    $j := p_j$
  return T

Complexity analysis. Throughout the course of the algorithm, we iterate over the sequence once, which is $O(n)$ operations. For each of these iterations, we iterate over some of our h different min-heaps, which in the worst case is at most $O(n)$. (On the average case it is a lot less, and I conjecture that even in the worst case it is in also less, since if h is very large, it means we often broke early on in the loop.) Additionally for each element in the sequence, we also perform a heap insertion, which is $O(\lg(n))$.

Finally to obtain the final result, we select the elements in the largest subsequence, which is h operations, and h is $O(n)$.

The complexity of the algorithm is therefore at most $O(n^2)$. (And according to my conjecture, it would be closer to $O(n \lg(n)))$

Regarding the space complexity, we keep two arrays p and l, each of length n, as well as h heaps, whose combined size is n (since each i is only inserted into one heap). Thus the algorithm uses $O(n)$ extra space.

3 Activity Selection Problem

We are given a set of activities $A = \{a_1, \ldots, a_n\}$, and a single resource. Each activity a_i has a start time s_i and a finish time f_i such that $f_i > s_i$. The time range of the activity a_i is the half-open interval $[s_i, f_i)$. We are also given the start time S and the finish time F of the resource. (If for activity i, $s_i < S$ or $f_i > F$ then this activity cannot use the resource.)

We shall denote by $A_{S,F}$ the subset of A such that all its activities can use the resource which starts at S and ends at F. More formally,

$$A_{S,F} = \{a_i \in A : s_i \geq S \wedge f_i \leq F\}$$

The goal is to find a set $C_{S,F} \subseteq A_{S,F}$ of mutually compatible activities such that the resource is used for a maximal amount of time, i.e. such that the value of $T_{S,F} = \sum_{a_i \in C_{S,F}} (f_i - s_i)$ is maximal.

3.1 A Greedy Algorithm

In the first lecture of the course a similar problem was presented along with a greedy algorithm which could solve it. In that problem the goal was to select as many compatible activities as possible, whereas in this problem we aim for maximal use of the resource.

3.2 A Counter-example to the Greedy Algorithm

To show that the greedy algorithm does not always return the correct result we will propose a counter-example. Suppose $S = 0$ and $F = 10$. Suppose the start times and finish times of the activities are as shown in table 2.

i	1	2	3
s_i	0	5	0
f_i	1	6	10

Table 2: Start times and finish times of activities

The greedy algorithm would first select the activity which finishes first, namely a_1, then would eliminate a_3 which is incompatible with a_1; then it would select a_2, which is the activity with the second smallest finish time. So it would return $\{a_1, a_2\}$, which uses the resource for $(1-0)+(6-5)=2$ units of time. However there is a solution which uses the resource for longer, namely $\{a_3\}$ which uses it for 10 time units.

3.3 The Optimal Substructure

If the solution to the problem is $C_{S,F}$, and it contains the activity a_x, then the subset of $C_{S,F}$ whose activities all finish before or at s_x — i.e. C_{S,s_x} — must be an optimal solution to a smaller problem where its resource has a finish time of s_x. Similarly, the subset of $C_{S,F}$ whose activities all start after or at f_x — i.e. $C_{f_x,F}$ — must be a solution to the smaller problem where the resource has a start time of f_x.

3.4 Recursive Formula

As defined above, $C_{S,F}$ denotes the set of compatible activities in $A_{S,F}$ such that $T_{S,F}$ is maximal (where $T_{S,F}$ is the amount of time that the activities in $C_{S,F}$ are running).

Additionally we will define the max of a collection of sets of activities as set which yields the largest $T_{S,F}$ (i.e. it uses the resource for longest).

$$C_{S,F} = \begin{cases} \phi & A_{S,F} = \phi \\ \max\limits_{a_i \in A_{S,F}} (C_{S,s_i} \cup \{a_i\} \cup C_{f_i,F}) & A_{S,F} \neq \phi \end{cases}$$

3.5 Algorithm

The following algorithm solves the problem, and should internally store each result of SelectActivities along with the given parameters, so that each time it is called we can search for a precomputed result if it exists.

3.6 Time Complexity

The complexity of the algorithm without using the memorisation technique is $O(2^n)$. However if the memorisation technique is applied, then there are at most

```
function SelectActivities(A, S, F)
    if A_{S,F} = φ then return φ
    C := φ
    for a_i ∈ A_{S,F} do
        C_0 := SelectActivities(A, S, s_i)
        C_0 := C_0 ∪ {a_i}
        C_0 := C_0 ∪ SelectActivities(A, f_i, F)
        C := Max(C, C_0)
    return C
```

n^2 distinct calls to SelectActivities, each of which is only computed once, so the complexity is $O(n^2)$.