Notes 14 for CS 170

1 Chain Matrix Multiplication

Suppose that you want to multiply four matrices $A \times B \times C \times D$ of dimensions 40×20, 20×300, 300×10, and 10×100, respectively. Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications (a good enough estimate of the running time).

To multiply these matrices as $(((A \times B) \times C) \times D)$ takes $40 \cdot 20 \cdot 300 + 40 \cdot 300 \cdot 10 + 40 \cdot 10 \cdot 100 = 380,000$. A more clever way would be to multiply them as $(A \times ((B \times C) \times D))$, with total cost $20 \cdot 300 \cdot 10 + 20 \cdot 10 \cdot 100 + 40 \cdot 20 \cdot 100 = 160,000$. An even better order would be $((A \times (B \times C)) \times D)$ with total cost $20 \cdot 300 \cdot 10 + 40 \cdot 20 \cdot 10 + 40 \cdot 10 \cdot 100 = 108,000$. Among the five possible orders (the five possible binary trees with four leaves) this latter method is the best.

How can we automatically pick the best among all possible orders for multiplying n given matrices? Exhaustively examining all binary trees is impractical: There are $C(n) = \frac{1}{n}\binom{2n-2}{n-1} \approx \frac{4^n}{n\sqrt{n}}$ such trees ($C(n)$ is called the *Catalan* number of n). Naturally enough, dynamic programming is the answer.

Suppose that the matrices are $A_1 \times A_2 \times \cdots \times A_n$, with dimensions, respectively, $m_0 \times m_1, m_1 \times m_2, \ldots m_{n-1} \times m_n$. Define a *subproblem* (remember, this is the most crucial and nontrivial step in the design of a dynamic programming algorithm; the rest is usually automatic) to be to multiply the matrices $A_i \times \cdots \times A_j$, and let $M(i,j)$ be the optimum number of multiplications for doing so. Naturally, $M(i,i) = 0$, since it takes no effort to multiply a chain consisting just of the i-th matrix. The recursive equation is

$$M(i,j) = \min_{i \le k < j} [M(i,k) + M(k+1,j) + m_{i-1} \cdot m_k \cdot m_j].$$

This equation defines the program and its complexity—$O(n^3)$.

```
for i := 1 to n do M(i,i) := 0
    for d := 1 to n − 1 do
        for i := 1 to n − d do
            j = i + d, M(i,j) = ∞, best(i,j) :=nil
            for k := i to j − 1 do
                if M(i,j) > M(i,k) + M(k+1,j) + m_{i−1} · m_k · m_j then
                    M(i,j) := M(i,k) + M(k+1,j) + m_{i−1} · m_k · m_j, best(i,j) := k
```

As usual, improvements are possible (in this case, down to $O(n \log n)$).

Run this algorithm in the simple example of four matrices given to verify that the claimed order is the best!

2 Knapsack

A burglar breaks into a house, and finds n objects that he may want to steal. Object i has weight v_i and costs about c_i dollars, and the burglar knows he cannot carry a weight larger than B. What is the set of objects of larger total value, subject to the constraint that their total weight is less than B? (In an alternative, legal, formulation, you are packing your knapsack for a hike, the knapsack has volume B, and there are i items that you may want to pack, item i has volume v_i and its "usefulness" is c_i; what is the set of items of larger total usefulness that will fit into the knapsack?)

Formally, the KNAPSACK problem is as follows:

Given: n items of "cost" $c_1, c_2, \ldots, c_n$ (positive integers), and of "volume" $v_1, v_2, \ldots, v_n$ (positive integers); a volume value B (for *b*ound).

Find a subset $S \subseteq \{1, \ldots, n\}$ of the items such that

$$\sum_{i \in S} v_i \leq B \tag{1}$$

and such that the total cost $\sum_{i \in S} c_i$ is maximized.

For reasons that will be clear later in the course, there is probably no algorithm for this problem that runs in time polynomial in the length of the input (which is about $n \log B$), however there is an algorithm that runs in time polynomial in n and B.

We want to solve this problem using dynamic programming, so we should think of how to reduce it to a smaller problem. Let us think of whether item n should or should not be in the optimal solution. If item n is not in the optimal solution, then the optimal solution is the optimal solution for the same problem but only with items $1, \ldots, n-1$; if item n is in the optimal solution, then it leaves $B - v_n$ units of volume for the other items, which means that the optimal solution contains item n, and then contains the optimal solution of the same problem on items $1, \ldots, n-1$ and volume bound $B - v_n$.

This recursion generates subproblems of the form "what is the optimal solution that uses only items $1, \ldots, i$ and volume B'"?

Our dynamic programming algorithm constructs a $n \times (B+1)$ table $M[\cdot, \cdot]$, where $M[k, B']$ contains the cost of an optimum solution for the instance that uses only a subset of the first k elements (of volume $v_1, \ldots, v_k$ and cost $c_1, \ldots, c_k$) and with volume B'. The recursive definition of the matrix is

- $M[1, B'] = c_1$ if $B' \geq v_1$; $M[1, B'] = 0$ otherwise.
- for every $k, B' \geq 1$,

$$M[k, B'] = \max\{M[k-1, B'] \ , \ M[k-1, B' - v_k] + c_k\}$$

The matrix can be filled in $O(Bn)$ time (constant time per entry), and, at the end, the cost of the optimal solution for our input is reported in $M[n, B]$.

To *construct* an optimum solution, we also build a Boolean matrix $C[\cdot,\cdot]$ of the same size.

For every i and B', $C[i,B'] = True$ if and only if there is an optimum solution that packs a subset of the first i items in volume B' so that item i *is* included in the solution.

Let us see an example. Consider an instance with 9 items and a bag of size 15. The costs and the volumes of the items are as follows:

Item	1	2	3	4	5	6	7	8	9
Cost	2	3	3	4	4	5	7	8	8
Volume	3	5	7	4	3	9	2	11	5

This is table M

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k=9$	0	0	7	7	7	11	11	15	15	15	19	19	19	21	23	23
$k=8$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k=7$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k=6$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k=5$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k=4$	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
$k=3$	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
$k=2$	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5	5
$k=1$	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2

And this is table C.

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k=9$	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1
$k=8$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k=7$	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$k=6$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k=5$	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1
$k=4$	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
$k=3$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
$k=2$	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
$k=1$	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

The solution is

Optimum: 23
Item 9 Cost 8 Volume 5
Item 7 Cost 7 Volume 2
Item 5 Cost 4 Volume 3
Item 4 Cost 4 Volume 4

The algorithm that fills the matrices is as follows.

```
algorithm Knapsack(B,n,c[],v[])
    for b = 0 to B
```

```
    if (v[1] ≤ b) then
      M[1, b] = c[1]; C[1, b] = true;
    else
      M[1, b] = 0; C[1, b] = false;
  for i = 2 to n
    for b = 0 to B
      if b ≥ v[i] and M[i − 1, b − v[i]] + c[i] > M[i − 1, b]
        M[i, b] = M[i − 1, b − v[i]] + c[i]; C[i, b] = true
      else
        M[i, b] = M[i − 1, b]; C[i, b] = false
```

3 All-Pairs-Shortest-Paths

Now we move on to a different type of dynamic programming algorithm, that is not on a sequence but a general graph: We are given a directed graph with edge weights, and wish to compute the *shortest path from every vertex to every other vertex.* This is also called the *all-pairs-shortest-paths* problem. Recall that the question only makes sense if there are no negative cycles, which we shall henceforth assume.

We already have an algorithm that we can apply to this problem: Bellman-Ford. Recall that Bellman-Ford computes the shortest paths from *one* vertex to all other vertices in $O(nm)$ time, where n is the number of vertices and m is the number of edges. Then we could simply run this algorithm n times, for each vertex in the graph, for a total cost of $O(n^2m)$. Since m varies from $n-1$ (for a connected graph) to n^2-n (for a fully connected graph), the cost varies from $O(n^3)$ to $O(n^4)$.

In this section we will present an algorithm based on dynamic programming that always runs in $O(n^3)$ time, independent of m. To keep the presentation simple, we will only show how to compute the lengths of the shortest paths, and leave their explicit construction as an exercise.

Here is how to construct a sequence of subproblems that can be solved easily using dynamic programming. We start by letting $T^0(i,j)$ be the length of the shortest *direct* path from i to j; this is the length of edge (i,j) if one exists, and ∞ otherwise. We set $T^0(i,i) = 0$ for all i.

Now we define problem $T^k(i,j)$ as the shortest path from i to j that is only allowed to use intermediate vertices numbered from 1 to k. For example, $T^1(i,j)$ is the shortest path among the following: $i \to j$ and $i \to 1 \to j$, and $T^2(i,j)$ is the shortest path among the following: $i \to j$, $i \to 1 \to j$, $i \to 2 \to j$, $i \to 1 \to 2 \to j$, and $i \to 2 \to 1 \to j$.

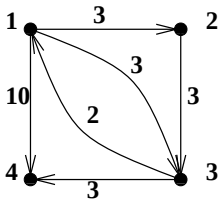
Next, we have to define $T^k(i,j)$ in terms of $T^{k-1}(.,.)$ in order to use dynamic programming. Consider a shortest path from i to j that uses vertices up to k. There are two cases: Either the path contains k, or it does not. If it does not, then $T^k(i,j) = T^{k-1}(i,j)$. If it does contain k, then it will only pass through k once (since it is shortest), and so the shortest path must go from i to k *via vertices* 1 *through* $k-1$, and then from k to j *via vertices* 1 *through* $k-1$. In other words $T^k(i,j) = T^{k-1}(i,k) + T^{k-1}(k,j)$. Putting these two cases together yields

$$T^k(i,j) = \min(T^{k-1}(i,j), T^{k-1}(i,k) + T^{k-1}(k,j)) \quad .$$

The algorithm, called the *Floyd-Warshall algorithm*, is now simple:

for $i, j = 1$ to n, $T^0(i,j) =$ length of edge (i,j) if it exists, 0 if $i = j$, and ∞ otherwise
for $k = 1$ to n
  for $i = 1$ to n
    for $j = 1$ to n
      $T^k(i,j) = \min(T^{k-1}(i,j), T^{k-1}(i,k) + T^{k-1}(k,j))$

Here is an example:



$$T^0 = \begin{bmatrix} 0 & 3 & 3 & 10 \\ \infty & 0 & 3 & \infty \\ 2 & \infty & 0 & 3 \\ \infty & \infty & \infty & 0 \end{bmatrix}, \quad T^1 = \begin{bmatrix} 0 & 3 & 3 & 10 \\ \infty & 0 & 3 & \infty \\ 2 & 5 & 0 & 3 \\ \infty & \infty & \infty & 0 \end{bmatrix}, \quad T^2 = T^1$$

$$T^3 = \begin{bmatrix} 0 & 3 & 3 & 6 \\ 5 & 0 & 3 & 6 \\ 2 & 5 & 0 & 3 \\ \infty & \infty & \infty & 0 \end{bmatrix}, \quad T^4 = T^3$$

Here is a simple variation on this problem. Suppose instead of computing the length of the shortest path between any pair of vertices, we simply wish to know whether a path exists. Given G, the graph $T(G)$ that has an edge from u to v if and only if G has a path from u to v is called the *transitive closure* of G. Here is an example:

$$G = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad T(G) = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

How do we compute $T(G)$? The solution is very similar to Floyd-Warshall. Indeed, we could just run Floyd-Warshall, since the length of the shortest path from u to v is less than ∞ if and only if there is some path from u to v. But here is a simpler solution: Let $\tilde{T}^k(i,j)$ be true if and only if there is a path from i to j using intermediate vertices from among $1, 2, ..., k$ only. The recursive equation defining $T^k(i,j)$ is

$$\tilde{T}^k(i,j) = \tilde{T}^{k-1}(i,j) \vee [\tilde{T}^{k-1}(i,k) \wedge \tilde{T}^{k-1}(k,j)]$$

This is nearly the same as the definition of $T^k(i,j)$ for Floyd-Warshall, with "$\vee$" replacing "min" and "$\wedge$" replacing "+". The dynamic programming algorithm is also gotten from Floyd-Warshall just by changing "min" to "$\vee$" and "+" to "$\wedge$", and initializing $\tilde{T}^0(i,j)$ to true if there is an edge (i,j) in G or $i = j$, and false otherwise.