CSC 373 H 1 Y — Summer 2007

University of Toronto — St. George Campus

Lecture Summary for Week 2

This summary is not a replacement for the lecture. If you miss a class, please arrange with a friend to take note for you.

1.4 The Matrix Multiplication Problem (continued)

The second attempt: Strassen's algorithm

Strassens algorithm for the Matrix Multiplication is based on the following observation:

$$C_{11} = P_5 + P_4 - P_2 + P_6 \qquad C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4 \qquad C_{22} = P_1 + P_5 - P_3 - P_7$$

where

$$P_1 = A_{11}(B_{12} - B_{22})$$
$$P_2 = (A_{11} + A_{12})B_{22}$$
$$P_3 = (A_{21} + A_{22})B_{11}$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$P_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

Exercise Verify that $C_{11}, \ldots, C_{22}$ can be computed as above.

The above formulas can be used to compute $A \times B$ recursively as follows:

MMult2(A, B)

1. If $n = 1$ Output $A \times B$
2. Else
3. Compute $A_{11}, B_{11}, \ldots, A_{22}, B_{22}$ Recursively.
4. Compute $P_1, \ldots, P_7$.
5. Compute $C_{11}, \ldots, C_{22}$.
6. Output C.
7. End If

The matrices A_{ij}, B_{ij} and P_k have size $\frac{n}{2} \times \frac{n}{2}$, so the addition and substraction involving them take time $\Theta(n^2)$. The running time for the algorithm satisfies:

$$T(n) = 7T(n) + \Theta(n^2)$$

Therefore $T(n) = \Theta(n^{\log_2 7})$ (around $\Theta(n^{2.81})$).

It has been shown that for multiplying 2×2 matrices, 7 multiplications are required.

In general, we can divide each $n \times n$ matrix A, B into k^2 smaller matrices of size $(n/k) \times (n/k)$ and try the same idea as in Strassen's algorithm, i.e., try to come up with formulas to compute the k^2 submatrices of $A \times B$ using only a matrix multiplications for matrices of size $(n/k) \times (n/k)$. Then the running time $T(n)$ will satisfy

$$T(n) = aT(n/k) + \Theta(n^2)$$

and $T(n) = \Theta(n^{\log_k(a)})$. In order to make improvement over Strassen's algorithm, we need $\log_k(a) < \log_2(7)$. This has been shown to be possible, e.g., for $k = 70, a = 143,640$.

1.5 Finding the Closest Pair of Points [Section 5.4]

Input A set P of n points $P = \{p_1, \dots, p_n\}$ where $p_i = (x_i, y_i)$.
Output A pair p_i, p_j of smallest distance.

In the one-dimensional case where all points lie on a line (says the x axis), the problem can be solved by sorting the point in increasing order of the x-coordinate (in time $\mathcal{O}(n\log(n))$, and find the two adjacent points with smallest distance (in time $\mathcal{O}(n)$).

The same idea does not work for the two-dimensional case, because two points can be far apart even though their x-coordinates differ very little.

There are $\frac{n(n-1)}{2}$ pairs of points, so the brute-force algorithm that computes the distances between all pairs will take time $\Theta(n^2)$.

Divide-and-Conquer: The idea

We will assume that the points have distinct x-coordinates. (Otherwise we can rotate the axes.)

1. Divide P into two halves Q, R by a vertical line ℓ
2. Compute recursively the closest pairs for Q and R
3. Compute the closest pair (p_i, p_j) where $p_i \in Q, p_j \in R$
4. Output the closest pair from step 2 and 3

In order to compute the two halves Q and R, one way is to sort the given points in increasing order of the x-coordinates. Since the set of input points is fixed, it is better to sort them only once before calling the recursive procedure (instead of sorting them at every recursive call).

Also, Q and R contain $n/2$ points each, so there are $(n/2)^2 = \Theta(n^2)$ pairs across Q and R. Suppose that in step 3 we simply check all of these $\Theta(n^2)$ pairs, then the running time $T(n)$ of the recursive procedure would satisfy

$$T(n) = 2T(n/2) + \Theta(n^2)$$

The Master Theorem gives us that $T(n) = \Theta(n^2)$. This is not assymptotically better than the brute-force algorithm above.

The fact that step 3 can be done in linear time is shown by M. I. Shamos and D. Hoey in the 70s. This can be shown as follows.

First, let δ be the smallest distance from step 2, i.e., any two points in Q and any two points in R have distance at least δ. In step 3, we are only looking for pairs that have distance less than δ. Therefore we can focus on points in Q and R that have distance less than δ from the dividing line ℓ.

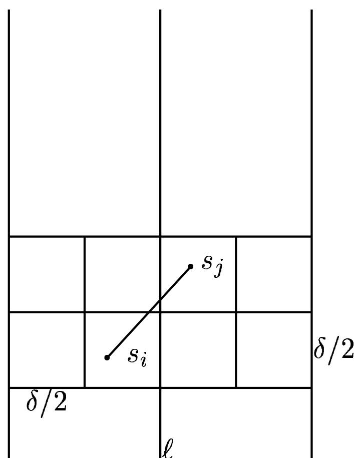
Thus, let S be the set of points in P that have distance $< \delta$ from ℓ. Suppose also that the points in S are sorted in increasing order of their y-coordinates:

$$s_1, s_2, \ldots, s_m$$

(where $m \leq n$). (This sorting can be done as a preprosessing step before running the recursive part of the algorithm, and takes time $\Theta(n \log(n))$.)

Consider two point s_i, s_j in S that have distance $d(s_i, s_j) < \delta$. We argue that the indices i, j cannot differ by more than a constant.

Without loss of generality, assume that $i < j$. Then s_i, s_j must lie in a $\delta \times 2\delta$ box B as depicted below.



Since the points in S are sorted in increasing order of their y-coordinates, any points s_k where $i < k < j$ must also lie in B.

On the other hand, each $\frac{\delta}{2} \times \frac{\delta}{2}$ square contains at most one point from Q or R (assuming that there are no point lying on the dividing line ℓ). This is because the distance of any two point in such a square is less than δ, and δ is the smallest distance of any two points in Q or in R. As a result, box B contains at most 8 points from S, i.e., there are at most 6 points s_k where $i < k < j$. Consequently, $j - i \leq 7$.

So step 3 can be done in linear time, since for each point s_i of S, we only need to compute the distance $d(s_i, s_j)$ where $i < j \leq i + 7$. (We need to compute at most $7n$ distances.)

The algorithm Closest-Pair(P)

1. $P_X \leftarrow P$ in increasing order of x-coordinate.
2. $P_Y \leftarrow P$ in increasing order of y-coordinate.
3. return Closest-Pair-Rec(P, P_X, P_Y).

Closest-Pair-Rec(P, P_X, P_Y)

1. If $|P| \leq 3$ then compute the closest pair in P by brute force.
2. Else
3. Compute line ℓ dividing P into Q, R
4. Compute Q_X, Q_Y and R_X, R_Y from P_X and P_Y.

5. $\langle q_0, q_1 \rangle$ = Closest-Pair-Rec(Q, Q_X, Q_Y)
6. $\langle r_0, r_1 \rangle$ = Closest-Pair-Rec(R, R_X, R_Y)
7. $\delta = min\{d(q_0, q_1), d(r_0, r_1)\}$
8. S = points in P within distance δ to ℓ in increasing order of y-coordinates
9. For each s_i in S,
10. Compute distances from s_i to s_j, where $i < j \leq i + 7$.
11. Let $\langle s', s'' \rangle$ be the closest pairs in S
12. End For
13. return the closest of $\langle q_0, q_1 \rangle$, $\langle r_0, r_1 \rangle$ and $\langle s', s'' \rangle$
14. End If

Exercise Verify that computing Q_X, Q_Y, R_X, R_Y (line 4) can be done in time $\mathcal{O}(n)$. Verify that computing S as on line 8 can also be done in time $\mathcal{O}(n)$.

Running time: The running time $T(n)$ for Closest-Pair-Rec satisfies

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Hence $T(n) = \mathcal{O}(n \log(n))$. The sorting (lines 1 and 2) in the procedure Closest-Pair takes time $\Theta(n \log(n))$. Altogether, the running time is $\Theta(n \log(n))$.

2 DYNAMIC PROGRAMMING [Chapter 6]

2.1 Longest Common Subsequence [CLRS 15.4]

Input Two sequences (arrays) $X[1 \ldots n]$, $Y[1 \ldots m]$.
Output A longest common subsequence of X, Y.

Example $X = TGACT, Y = GTCGAT$. The longest common subsequence is $TGAT$ (length 4).

Suppose that $n \leq m$, the brute-force algorithm (that compares all possible pairs of subsequences of X and Y) takes time

$$\sum_{k=1}^{n} \frac{n!}{k!(n-k)!} \frac{m!}{k!(m-k)!}$$

which is at least exponential in n.

For a recursive algorithm, let $LCS(i, j)$ denote the longest common subsequence of $X[1 \ldots i]$ and $Y[1 \ldots j]$. Then

$$LCS(n, m) = \begin{cases} LCS(n-1, m-1) \circ X[n] & \text{if } X[n] = Y[m] \\ \text{longer of } \{LCS(n-1, m), LCS(n, m-1)\} & \text{otherwise} \end{cases}$$

Using the above relation, we have:

A Recursive Program
LCS(X,Y,n,m):

1. If $n = 0$ or $m = 0$: return $\varnothing$
2. If $X[n] = Y[m]$ return $LCS(X, Y, n-1, m-1) \circ X[n]$
3. Else
4. $\quad L_1 = LCS(X, Y, n-1, m)$
5. $\quad L_2 = LCS(X, Y, n, m-1)$
6. $\quad$ If $|L_1| > |L_2|$
7. $\quad\quad$ return L_1
8. $\quad$ Else
9. $\quad\quad$ return L_2
10. $\quad$ End If
11. End If

In the worst case (i.e., all characters in X are different from the characters in Y) $LCS(X, Y, n, m)$ always makes two recursive calls $LCS(X, Y, n-1, m)$ and $LCS(X, Y, n, m-1)$. So the running time of LCS is still at least exponential in n (for $n \leq m$).

There are only $n \times m$ subproblems to be solved (i.e. finding the longest common subsequence of $X[1 \ldots i]$ and $Y[1 \ldots j]$). The fact that the algorithm LCS above takes exponential time is essentially due to the fact that it recomputes these longest common subsequences many times. To avoid repeated computations, one way is to solve these subproblems only once and store their solutions. This suggests a bottom up approach.

An Improvement

LCS2(X,Y,n,m):

1. S: $n \times m$ array of sequences of length at most n % For storing solutions.
2. For $j = 1$ to m do $S[0, j] \leftarrow \varnothing$
3. For $i = 1$ to n do $S[i, 0] \leftarrow \varnothing$
4. For $i = 1$ to n do
5. $\quad$ For $j = 1$ to m do
6. $\quad\quad$ If $X[i] = Y[j]$ do $S[i, j] \leftarrow S[i-1, j-1] \circ X[i]$
7. $\quad\quad$ Else $S[i, j] \leftarrow$ longer of $\{S[i-1, j], S[i, j-1]\}$
8. $\quad\quad$ End If
9. $\quad$ End For
10. End For
11. Return $S[n, m]$

The running time is no longer exponential. In the main part, there are $n \times m$ loops. Suppose that the copy operations on lines 6 and 7 takes time n, then the running time is $\mathcal{O}(n \times m \times n)$.

Note that storing the whole solution for each subproblem is not necessary. What we really need is some information that allows us to construct the longest common subsequence of X and Y. Storing the whole solutions for the subproblems clearly suffices for this purpose, but we will be able to manage with storing less information (thus saving on the space needed). This will be discussed next week.