CMPSCI 611 Advanced Algorithms **Fall 2000**

Lecture 2: September 12

Lecturer: Micah Adler *Scribes: Gary Holness and TJ Brunette*

2.1 Matrix Multiplication

Input: Two $n \times n$ matrices A, B with entries from any ring (e.g., the real numbers)

Output: $C = A \times B$

We could do this by computing each term one at a time requiring n operations for each term resulting in $O(n^3)$ operations. The thing to keep in mind is that this really isn't an n^3 algorithm. Because matrices are $n \times n$, the input size is n^2. We will look at another approach, namely Strassen's amazing algorithm [S69], which will do better than n^3. Strassen's is a divide and conquer algorithm.

2.1.1 A Divide and Conquer Algorithm for Matrix Multiplication

The Idea: subdivide the matrix into sub-matrices, looking at it as 4 quadrants of sub-matrices

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

We will use this representation as the basis of a divide and conquer algorithm. We treat these as 2×2 matrices and write down what the resulting product matrix would be.

$$\mathbf{C} = \begin{pmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{22} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{22} \end{pmatrix} \tag{2.1}$$

To verify, let's look at the element in the first row, first column of the matrix C.

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} \tag{2.2}$$

This is a valid expression for the matrix C which we have expressed in terms of smaller multiplications.

What is the running time?

$$\begin{aligned} T(n) &= 8T(\frac{n}{2}) + \Theta(n^2) \qquad \text{for } n > 1 \\ T(1) &= \Theta(1) \end{aligned} \tag{2.3}$$

We've divided into 8 sub-matrices, each of size $\frac{n}{2}$. The amount of time to combine is $\Theta(n^2)$ because we are adding $\frac{n}{2} \times \frac{n}{2}$ sub-matrices. Note that $\alpha = 2$ and $\beta = \log_2 8 = 3$, so by master method, this is a $\Theta(n^3)$ algorithm. We haven't improved on the naive algorithm. We have only reordered the computation. As we did last time with the multiplication of two n-bit numbers, we can reduce the number of subproblems. So, we can improve on the n^3.

2.1.2 Strassen's algorithm

Previously we had 8 recursive subproblems. We will rewrite them and express C in terms of 7 subproblems.

$$\begin{aligned}
P_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22}) \times B_{11} \\
P_3 &= A_{11} \times (B_{12} - B_{22}) \\
P_4 &= A_{22} \times (-B_{11} + B_{21}) \\
P_5 &= (A_{11} + A_{12}) \times B_{22} \\
P_6 &= (-A_{11} + A_{21}) \times (B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22})
\end{aligned}$$

Each product requires some amount of addition, subtraction and some multiplies. We can compute C by combining these recursive products as follows ...

$$\mathbf{C} = \begin{pmatrix} P_1 + P_4 - P_5 + P_7 & P_3 + P_5 \\ P_2 + P_4 & P_1 - P_2 + P_3 + P_6 \end{pmatrix}$$

Lets look at $P_2 + P_4$ to demonstrate that this works.

$$\begin{aligned}
P_2 + P_4 &= ((A_{21} + A_{22}) \times B_{11}) + (A_{22} \times (-B_{11} + B_{21})) \\
P_2 + P_4 &= A_{21} \times B_{11} + A_{22} \times B_{11} - A_{22} \times B_{11} + A_{22} \times B_{21} \\
P_2 + P_4 &= A_{21} \times B_{11} + A_{22} \times B_{21}
\end{aligned}$$

The run time of this algorithm is:

$$\begin{aligned}
T(n) &= 7T(\frac{n}{2}) + \Theta(n^2) \qquad n > 1 \qquad (2.4) \\
T(1) &= \Theta(1) \\
\alpha &= 2 \\
\beta &= \log_2 7 \approx 2.81 \\
T(n) &\approx \Theta(n^{2.81})
\end{aligned}$$

Here we are still just doing a constant number of additions and subtractions, but this constant is bigger and we still have a $\Theta(n^2)$ term.

In equation 2.1, there are only 4 additions and subtractions. Looking at Strassen's algorithm, counting all the additions and subtractions, there are 18. This means that, in Strassen's, there are bigger constants. Specifically, the Θ term in equation 2.4 hides a bigger constant than the Θ term in equation 2.3. If you go and do all the math, Strassen's algorithm is going to be better for $n \geq 100$. Asymptotically, Strassen's will be better, but the asymptotic behavior doesn't take over until fairly large values of n. So, if you were writing a program to multiply very large matrices, you would keep subdividing the matrix using Strassen's algorithm. Once you reached a value of n of about 100, you would then keep recursing with the more straightforward approach.

It turns out that Strassen's is not the fastest known algorithm for matrix multiplication. You can think of Strassen's algorithm as multiplying two 2×2 matrices using 7 multiplies. Pan extended this [P84]. Instead of considering 2×2 sub-multiplies (dividing things into recursive subproblems $\frac{1}{4}$th the size), he divided them into much smaller pieces.

Pan came up with a way to multiply two:

68×68	matrices using	$132,464$	*multiplies*
70×70	matrices using	$143,640$	*multiplies*
72×72	matrices using	$155,424$	*multiplies*

We leave as an exercise, which of these would lead to the asymptotically fastest algorithm for a divide and conquer approach. Currently, the asymptotically fastest known algorithm for matrix multiplication is $O(n^{2.376})$ [CW90]. This uses different techniques from divide and conquer. As far as a lower bound is concerned, can we do better than this? We know the lower bound on any matrix multiplication is $\Omega(n^2)$ because we must look at every cell of an $n \times n$ matrix. An interesting open problem is "what is the true lower bound?"

2.2 Closest Points in a Plane

Lets say we have a set of points in a plane and want to compute the 2 points which are closest together.

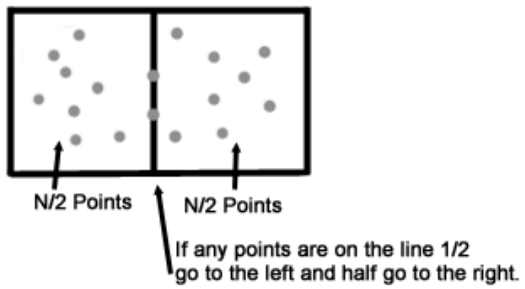

Figure 2.1:

Given: A set P of n points in the plane.

Let: $d(p, q)$ be the distance between points p and q.

Determine: The pair $p, q \in P$ $(p \neq q)$ such that $d(p, q)$ is minimal.

Actually: we will just compute the minimum distance instead of the pair of points which achieve it, i.e., min $d(p, q)$ where $p, q \in P$ and $p \neq q$.

Naive Algorithm: try all pairs. This requires time $\Theta(n^2)$.

Can we do better? Yes. We will again use the divide and conquer approach.

2.2.1 Divide and Conquer Approach to Closest Points in a Plane

1. If $n < 3$, try all pairs (this takes constant time because there is only at most 1 pair to try)

2. Otherwise divide P into two halves with a vertical line X such that $\frac{n}{2}$ points are on the left side and right side of it. We must be careful that we don't have points which lie on the line. We will make sure to assign points on the line so that, we have equal numbers assigned to each side. We get $\frac{n}{2}$ points in P_L and P_R where P_L are points on the left and P_R are points on the right.

3. Recursively find the
δ_L: minimum distance between points in P_L
δ_R: minimum distance between points in P_R
The answer is not just just take the minimum of these two values, we must also consider the distances between points that cross the dividing line, so

4. (Combining) Test every pair which crosses the dividing line
$\delta_M = \min d(p, q)$ where $p \in P_L$ and $q \in P_R$.
This step requires $\Theta(n^2)$ time.

5. Return $\min(\delta_L, \delta_R, \delta_M)$

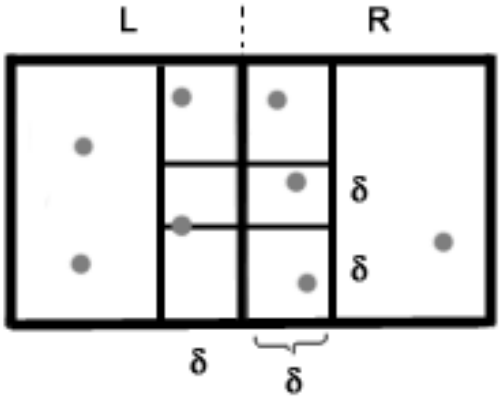

Figure 2.2:

Unfortunately the algorithm as stated is clearly $\Theta(n^2)$, since $T(n) = 2T(n/2) + \Theta(n^2)$, and $T(1) = 1$. We can improve on this. The key here is that when we go to step 4, we have already solved the recursive subproblems. We know that the value returned must be no greater than $\delta = \min(\delta_L, \delta_R)$. So, we will use this information to be more efficient about how we test the pairs that cross the dividing line. For each point p we only compare to points on the other side of the dividing line that are within distance δ of p both horizontally and vertically. If the distance between p and q is less than δ then q will be in the rectangle.

4a. Let P_M be the set of points within δ of the dividing line.

4b. For each point $p \in P_M$ that is to the left of the dividing line, check the distance to all points q in the following rectangle (right is similar):

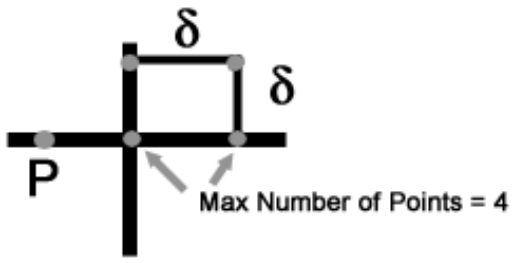

Figure 2.3:

Then δ_M is the minimum distance between any such pair of points. What is the maximum number of points that can appear in the box to be checked? We know that the box is $\delta \times \delta$, and the points must be at least distance δ from each other. The answer is thus obviously four. This means that the total number of pair-wise comparisons for step 4 is $\Theta(n)$. If we now assume that there is no further work we derive time as follows:

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + \Theta(n) \\ T(1) &= \Theta(1) \\ T(n) &= \Theta(n \log n) \end{aligned}$$

We still need to take into consideration that we must be able to

- Split P into P_L and P_R, and
- find the points in the rectangle for each point.

How do we do this?

- Sort the points by their x-value. This allows us to find P_L, P_R, P_M in linear time.
- Sort the points by their y-value. We can find points in the rectangle for a given point in linear time.

We can sort the points at each step, but that would require more than $\Theta(n)$ work. The key is that we can sort the points once at the start of the algorithm and then extract sorted sub-lists in time $\Theta(n)$. So, in linear time search sorted sub-lists for P_L, P_R, P_M. So the total time for the algorithm is $\Theta(n \log n)$.

2.3 Fast Fourier Transform (FFTs)

In the following part of the lecture we will describe FFTs from the point of view of an application, multiplying polynomials.

Application: Multiplying two degree N polynomials

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \\ B(x) &= b_0 + b_1 x + b_2 x^2 + \ldots + b_n x^n \\ C(x) &= A(x) \cdot B(x) \end{aligned}$$

Adding together polynomials: We can add two polynomials by adding similar terms:

$$A(x) + B(X) = a_0 + b_0 + (a_1 + b_1)x + (a_2 + b_2)x^2 \ldots$$

This is clearly $\Theta(n)$.

Multiplying polynomials: Multiplying is not quite as simple as addition. The obvious algorithm is to multiply each pair of terms

$$A(x) \cdot B(x) = (a_0 \times b_0 + a_0 \times b_1 x \ldots) + (a_1 x \times b_0 + a_1 x \times b_1 x \ldots) + \ldots \tag{2.5}$$

The obvious algorithm uses time $\Theta(n^2)$.

The above polynomials are in coefficient representation. However we could represent the polynomials using a point-value representation where we evaluate the polynomial at any distinct $n+1$ points as shown below.

$$\begin{aligned} A(x) &= \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\} \\ \text{where } y_i &= A(x_i) \end{aligned}$$

Claim: point-value representations with $n+1$ distinct points define a unique n-degree polynomial [CLR, pp 779–780].

If you have $n+1$ points each point value pair gives you a linear equation in the coefficient representation. Thus you get $n+1$ linear equation with $n+1$ unknowns and you can show there is a unique solution for each unknown.

Addition: use the same x values:

$$\begin{aligned} A(x) &= \{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\} \\ B(x) &= \{(x_0, y_0'), (x_1, y_1'), \ldots, (x_n, y_n')\} \end{aligned}$$

Add corresponding terms using point value representation:

$$A(x) + B(x) = \{(x_0, y_0 + y_0'), (x_1, y_1 + y_1'), \ldots, (x_n, y_n)\}$$

Multiplication: $A(x) \times B(x)$ In point value representation is the same as addition except that you multiply corresponding terms instead of adding them.

Idea: You need to have more terms to multiply two n-degree polynomials in order to return a degree $2n$ polynomial. So start with $2n+1$ point polynomials then multiply corresponding terms.

Outline: Below is an outline of an algorithm we are going to use for multiplying two polynomials.

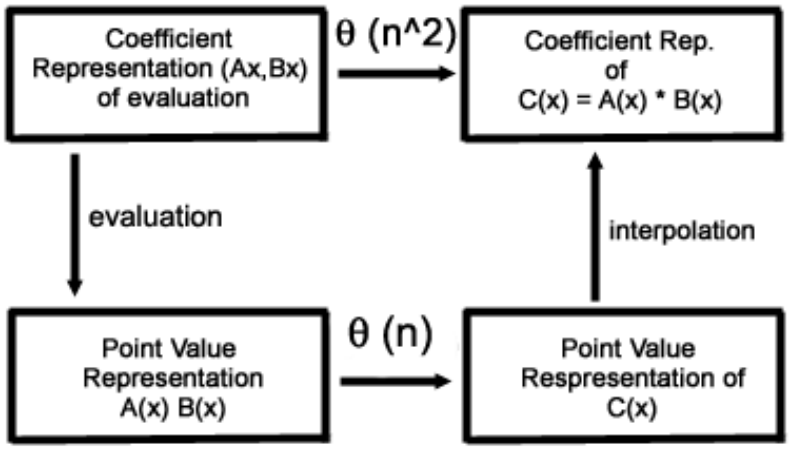

Figure 2.4:

The FFT approach will take the path in the above figure that goes through evaluation and interpolation.

Evaluating a polynomial of degree $n-1$ at n points each evaluated in n time. The obvious algorithm: $\Theta(n^2)$ time but by using a FFT that can be reduced.

Evaluation: evaluating a polynomial at many points,

Interpolation: attempting to interpolate a coefficient representation of a polynomial from a point value representation.

Both evaluation and interpolation can both be done in $\Theta(n \log n)$ time.

FFT: Evaluate a degree $n-1$ polynomial at n special points in $\Theta(n \log n)$ time

n **special points:** we'll use the n complex nth roots of unity, i.e., the n solutions to $x^n = 1$

Roots of unity : are n equally spaced points around a unit circle, with the 0th root lying on the x-axis.

ω_n**:** principal n^{th} root of unity first counter clockwise around the circle starting at 1

Useful properties:

$$\omega_n^{k+n} = \omega_n^k$$
$$(\omega_n)^{n/2} = -1$$

If n is even the squares of the n complex nth roots of unity are the $\frac{n}{2}$ complex $(\frac{n}{2})^{nd}$ roots of unity repeated twice. This is known as the Halving Lemma. If we look at the 8 roots of unity shown in the above graph and square them we get 4 roots of unity each repeated twice.

References

CW90 D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *Journal of Symbolic Computation*, Volume 9, Number 3, March 1990, pp. 251-280.

S69 V. Strassen, Gaussian Elimination Is Not Optimal, *Numerische Mathematik* **13**, 1969, pp. 354–356.

P84 V. Pan, *How To Multiply Matrices Faster*, Springer-Verlag, Lecture Notes in Computer Science Vol. 179, 1984.

CLR T. Cormen and C. Leiserson and R. Rivest, Introduction to Algorithms, MIT Press, 1990