1
Triangle Counting and Matrix Multiplication

In the first lecture of the course, we will see how the seemingly simple problem of counting the number of triangles in a graph gives rise to interesting algorithmic ideas, and some unexpected connections.

1.1 Counting Triangles in a Graph

Given an undirected graph $G = (V, E)$, the ***triangle counting problem*** asks for the number of triangles in this graph. For example, the graph on the right contains 4 triangles. This problem arises in the analysis of social networks: the density of triangles in the graph (i.e., the number of triangles divided by $\binom{n}{3}$) is called the *global clustering coefficient* of a network.

How fast can we solve this triangle-counting problem? Clearly, this answer depends on the size of the graph. In ***worst-case analysis***, we fix some parameters of the graph: say, the number of nodes $|V|$ (typically denoted by n) and/or the number of edges $|E|$ (denoted by m), and gives a worst-case bound on the running-time of the algorithm for any graph with at most n nodes and m edges. Let's see some examples of such algorithms and their analyses.

In this course, when dealing with graphs, we will use $n = |V|$ and $m = |E|$, typically without comment.

1.1.1 Attempt I: The Naïve Algorithm

A first naïve algorithm is to enumerate over all triples $\{u, v, w\}$ of the vertex set V and count how many of these form a triangle. (We divide by 6 to compensate for the overcounting we count each triangle $3! = 6$ times.) There are $\binom{n}{3}$ such triples, and if we assume that checking the presence of an edge can be done in constant time, this takes $O(n^3)$ time.[1] This algorithm's runtime does not depend on how dense the graph is, how many edges it has, it always takes the same amount of time—which suggests a way to improve on it.

[1] This is a good time to ask how the graph is represented. For this algorithm, assume we are given the input as an *adjacency matrix* $A \in \{0,1\}^{n \times n}$, where $A_{uv} = 1$ if the edge $\{u, v\}$ is present, and 0 otherwise. A different representation would be as an *adjacency list*, where each vertex has a linked list giving its adjacent edges.

Algorithm 1: The Naïve Algorithm for Counting Triangles

```
1.1 C ← 0
1.2 for u = 1...n do
1.3 |  for v = i...n do
1.4 |  |  for w = j...n do
1.5 |  |  |  if u, v, w is a triangle then
1.6 |  |  |  |  C ← C + 1
1.7 return C/6.
```

1.1.2 *Attempt I: The Edge-Scan Algorithm*

A different algorithm is to enumerate over all vertices $v \in V$, then to enumerate over all pairs of edges vu, vw incident to v, and check if these edges are part of a triangle (i.e., if $\{u, w\}$ is an edge of the graph or not).

Algorithm 2: Edge-Scan for Counting Triangles

```
2.1 C ← 0
2.2 for v = 1...n do
2.3 |  for each pair of edges vu, vw incident to v do
2.4 |  |  if u, v, w is a triangle then
2.5 |  |  |  C ← C + 1
2.6 return C/3.
```

If d_v is the ***degree*** of the vertex v—the number of edges incident to v—there are $\binom{d_v}{2}$ such pairs of edges to be checked for vertex v. Hence the total number of pairs of edges is

$$\sum_v \binom{d_v}{2} \leq \sum_v d_v^2 \leq d_{\max} \sum_v d_v = d_{\max} \cdot 2m \leq O(nm).$$

Here we used the fact that the sum of the degrees equals twice the number of edges of the graph—each edge is counted twice in the sum, one from each end. (This is sometimes called the ***Handshake lemma***.)

In the calculations above, observe that each step of the analysis potentially weakens the quantitive bound, makes it less nuanced but more universal, and therefore easier to write and interpret. The final bound is only in terms of the number of nodes n and edges m, as we wanted. Moreover, since each graph has at most $\binom{n}{2}$ edges, we have $m = O(n^2)$, so the $O(mn)$ bound clearly subsumes the previous $O(n^3)$ bound. Moreover, our analysis is "tight" for the algorithm: running the algorithm on the star graph on the right takes $O(n^2)$

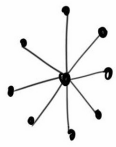

Figure 1.1: A Star graph.

time, but since $m = n - 1 = O(n)$ in this case, this $O(n^2)$ matches the $O(mn)$ upper bound we proved.

Exercise 1.1. Can we get a matching example for every choice of n and $m \leq \binom{n}{2}$?

Let's improve Algorithm 2 further in two ways: the first is directly inspired by star graph above, whereas the second is surprising and will lead to other connections.

1.1.3 Attempt III: The Low-Degree Advantage

For the first improvement, observe that Algorithm 2 above count each triangle multiple times, once from each of its vertices. How about cutting down the search space? Here's one way: order the nodes in increasing order of their degree, so that

$$d_1 \leq d_2 \leq d_3 \leq \cdots \leq d_{n-1} \leq d_n.$$

Now, when looking for triangles from node v, only consider pairs of edges going from v to higher numbered nodes. (Think about what this would do on the star graph example above: it would look over all the leaves, which have unit degree, and so determine that there are no triangles in the graph in $O(n)$ time.) You could try to prove the following (slightly tricky) exercise either now, else you will probably see it in HW#1.

Exercise 1.2. Show that the runtime of this algorithm is $O(m^{1.5})$.

1.1.4 Attempt IV: A New Hope

A powerful way to get better algorithms, and indeed to solve problems in general, is to ***change the representation***!

Suppose we write the graph as the adjacency matrix $A \in \{0,1\}^{n \times n}$. Since we have a matrix (with entries zeros and ones) we can think about the rich set of operations we are allowed to perform with matrices. In later lectures, we will use many of the advanced properties of matrices, but today let's look at the most basic of operations. *We are allowed to add and multiply matrices.* If you multiply the matrix A with itself, the entry

$$(A \cdot A)_{ij} = \sum_{k=1}^{n} A_{ik} A_{kj}.$$

We get a contribution of one for each index k where there is an edge ik as well as an edge kj—in other words, the $(i,j)^{th}$ entry of A^2 counts the number of two-hop paths from i to j! Since a triangle $\{i,j,k\}$

Algorithm 3: Counting Triangles by Matrix Squaring

```
3.1 C ← 0
3.2 compute A^2
3.3 for i = 1...n do
3.4 |   for j = 1...n do
3.5 |   |   C ← C + A_ij · (A^2)_ij
3.6 return C/3.
```

means we have a two-hop path ik, kj as well as an edge ij, we can use matrix A and its square A^2 to count the number of triangles:

Clearly, we can do all steps except Line 3.2 in $O(n^2)$ time, so the question is now: how fast can compute the square of a matrix? In general, given two $n \times n$ matrices A, B, how fast can be multiply them? If we denote this time by $T_{MM}(n)$, then we can clearly square a matrix (i.e., compute the product of A with itself) in this time, and hence count the number of triangles in time

$$O(n^2) + T_{MM}(n).$$

We now focus on the question: how low can $T_{MM}(n)$ be?

1.2 *The Complexity of Matrix Multiplication*

As always, it is useful to start off with the naïve algorithm for a problem. And in this case, this comes directly from the definition:

$$(AB)_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

So computing each entry of AB takes n multiplies, and $n-1$ additions. That's $O(n)$ time. And there are n^2 entries, so we get $O(n^3)$ time to multiply two $n \times n$ matrices.

Can we do better? The answer, surprisingly, is ***Yes***. This sub-cubic algorithm for matrix multiplication was a surprise when it was first discovered, and still seems surprising to me. How else could you multiply two matrices faster? *Even the definition takes $O(n^3)$ steps to implement.* This, by the way, is a dangerous way to reason about things. There is no reason that the definition would suggest the right way to implement something. Since the definition suggests an algorithm that uses $O(n^3)$ steps, and we want to do faster, we should step away from the definition, and try to see other ways to solve the problem.

For the rest of this section, we assume that the entries of the matrices are numbers belonging to some ***ring***, and we can add and multiply them in constant time.

Beware of thinking of the definition of a problem as the only way, or even the "right" way to solve it. E.g., the definition of sorting suggests an algorithm that takes $n!$ time: try all orderings until you find one where each element is smaller than the one following it. But we can sort n numbers in $O(n \log n)$ time, using MergeSort or (randomized) QuickSort.

A ***ring*** is a set of numbers, for which we have a well-defined notion of addition and multiplication: we also have numbers 0 and 1 such that $a + 0 = a$ and $a \times 1 = a$. Finally, we also have, each number a has an additive inverse $(-a)$ such that $a + (-a) = 0$. For instance, the set of integers with the usual notions of sum and product form a ring.

1.2.1 *Approach I: Divide and Conquer*

One approach is the simple and powerful divide-and-conquer paradigm: can we implement multiply two $n \times n$ matrices using fewer than 8 multiplications of $n/2 \times n/2$ matrices? For simplicity, assume that n is even, and hence view the product AB

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

A little thought shows that the answer is

$$\begin{bmatrix} A_{11}B_{11} + A_{12}B_{22} & A_{11}B_{21} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{22} & A_{21}B_{21} + A_{22}B_{22} \end{bmatrix} \tag{1.1}$$

Now the strategy is clear: we can use 8 products of half-sized matrices to compute the products in (1.1), and then use 8 more additions to get the answer. Finally, if $n = 2$, then the half-sized matrices can be multiplied in constant time.

As we will see, sums are cheap, and hence we would be happy to use any larger constant number of additions.

Great! How much time does this take? If $T(n)$ is the time to multiply $n \times n$ matrices, then

$$T(n) = 8\,T(n/2) + 8n^2.$$

In fact, to show the differing role of these two factors of 8, let's write this as:

$$T(n) = 8\,T(n/2) + c\,n^2.$$

The best way to solve this is to "unroll" it: use the definition to write $T(n/2)$ in terms of $T(n/4)$:

$$\begin{aligned} T(n) &= 8\,(8T(n/4) + c(n/2)^2) + cn^2 \\ &= 8^2\,T(n/4) + cn^2(1+2) \end{aligned}$$

And in general, we can go on to get

$$= 8^i\,T(n/2^i) + cn^2(1 + 2 + \cdots + 2^{i-1}).$$

So if we go until $i = \log_2 n$, we get

$$= 8^{\log_2 n}\,T(1) + cn^2(1 + 2 + \cdots + 2^{\log_2 n - 1}).$$

Great. Let's see:

$$8^{\log_2 n} = n^{\log_2 8} = n^3.$$

And

$$(1 + 2 + \ldots + 2^{i-1}) = (2^{i-1} - 1)$$

which again gives

$$cn^2(1 + 2 + \cdots + 2^{\log_2 n - 1}) = cn^2(2^{\log_2 n} - 1) = cn^2(n-1).$$

Hmm, both the terms give us n^3 again. That's a disappointment: we did all this work, and sadly did not get much better than the cubic runtime we'd got earlier.

It turns out that we're close, as we will soon see. But before we see how to use this technology to actually do better, observe that improving the number of additions would only have changed the constant term. On the other hand, changing the number of multiplications from 8 to 7 would really make a difference, since it would change the $8^{\log_2 n}$ to $7^{\log_2 n} = n^{\log_2 7}$.

Indeed, it's worth observing this carefully: given the recursive structure, changing 8 recursive calls to 7 changes the recursion tree to have far fewer leaves. And that's where most of the work in this recursive algorithm happens, so the improvement is tremendous.

1.2.2 *Approach II: Conquer Smarter*

When Volker Strassen was trying to solve this problem, he realized he should start at the bottom, at the case $n = 2$: *could he multiply two* 2×2 *matrices in less than* 8 *scalar multiplications (plus a small constant number of additions)?* I.e.,

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad = \quad ??$$

If he could, and if this solution did not use very much about scalar multiplication, then maybe he could use the same idea in the inductive step.

He used another great algorithmic idea first: he tried to prove a lower bound. ***He tried to show that his goal was not possible.*** And indeed, he soon showed that six multiplications were not enough, no matter how smartly one used them. But he could not rule out the possibility that seven multiplies sufficed.

So then he returned to the question of getting an algorithm. Clearly, he could not just multiply the eight numbers that formed the input pairwise and sum them up, that evidently would fail. So he started to combine them together, and soon realized he could do the 2×2 matrix multiplication with 7 scalar multiplications! If we define

Strassen's original construction was cosmetically different; we present a more symmetric solution with a nicer "geometric" structure.

$$S_1 = (a_{11} + a_{21})(b_{11} + b_{12})$$
$$S_2 = (a_{12} + a_{22})(b_{21} + b_{22})$$
$$S_3 = (a_{11} - a_{22})(b_{11} + b_{22})$$
$$S_4 = a_{11}\,(b_{12} - b_{22})$$
$$S_5 = (a_{21} + a_{22})\,b_{11}$$
$$S_6 = (a_{11} + a_{12})\,b_{22}$$
$$S_7 = a_{22}\,(b_{21} - b_{11})$$

then a little algebra shows that the product AB is

$$\begin{bmatrix} S_2 + S_3 - S_6 - S_7 & S_4 + S_6 \\ S_5 + S_7 & S_1 - S_3 - S_4 - S_5 \end{bmatrix} \tag{1.2}$$

Moreover, this uses nothing special about scalar multiplication: this same recursive formula works to solve the general $n \times n$ matrix multiplication using 7 multiplications of matrices of half the size (plus 18 matrix additions). Hence, the total runtime of multiplying two $n \times n$ matrices can be can be written using the recursion

$$T(n) = 7\,T(n/2) + 18n^2.$$

And the same simple arguments we used to solve the recursion above show:

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81}).$$

Strassen had broken the n^3 barrier for matrix multiplication!

Again, the constant hidden in the big-Oh is 18.

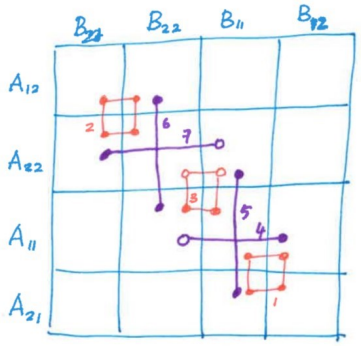

Figure 1.2: Strassen's Multiplications.

1.2.3 *A Visual Explanation of Strassen's Formula*⋆

There have been several attempts at making the calculations above more transparent and less mysterious. One that I particularly like is the a geometric depiction of the algorithm (shown in the figure) that I learned from Mike Paterson: each purple edge and red cycle represents one multiplication, where empty circles denote negations.

1.2.4 *A Recap, and Future Work*

To me, the main take-away from Strassen's approach is that divide-and-conquer works: one can multiply half-sized matrices using 7 multiplies, and that makes the difference. The exact structure of these multiplies is mysterious (and eventually has some deep connections), but these are not important at the first cut. For example, faced with this question, you could try to write a program to enumerate over all small combinations of multiplications and additions to find this solution. (Using the symmetry of the problem can help you cut the search space further.)

Strassen's approach was a real breakthrough: a series of subsequent works improved the exponent of n down to 2.38 (Coppersmith and Winograd, 1989). Since then, the improvements have slowed down quite a bit. Moreover, the improvements beyond Strassen's are important theoretically, but they are complicated and the constants in the big-Oh are larger, making them less practical. Strassen's algorithm, however, is something that can be used practically, especially in large parallel and distributed applications; see, e.g., papers and discussions here.

The best current result is due to Josh Alman and Virginia Vassilevska Williams (a CMU alumna).

1.3 Conclusions

Here are some take-aways from this lecture:

1. The definition of a problem may suggest an algorithm, but it may not be the "right" algorithm. This happened for both the triangle counting problem, and then later for matrix multiplication.
2. Similary, choose the right representation for the data, don't always use the representation given as input. For example, choosing the adjacency matrix representation instead of, say, the adjacency list representation, allowed us to get a faster algorithm.
3. If you are stuck proving upper bounds (algorithms), try to prove a lower bound. You may succeed! Or else, it may nudge you in the right direction ("why am I not being able to prove a lower bound?")
4. For me, the real moral of Strassen's algorithm is not the magic way of reducing the number of multiplies to 7, but the fact that divide-and-conquer allows us to translate this constant-factor reduction from 8 to 7 multiplies into an asymptotic improvement, using the power of recursion. In fact, the second most important take-away is that the best way of computing $X + Y$ may not be to compute X and Y separately and then to sum them, but instead to compute them some other way.

We will use these ideas repeatedly in the coming weeks. As for open questions, here are two immediate ones:

1. Give an algorithm to find a triangle in a graph (if one exists) in linear time $O(m)$? Any runtime faster than $O(m^{1.5})$ would be pretty good!
2. Multiply matrices in time faster than $n^{2.38}$, which is the current world record.