# CS 124 Programming Assignment 1: Spring 2022

**Your name:** Tram B. Nham
**Collaborators:**
**No. of late days used on previous psets:** 20
**No. of late days used after including this pset:** 23

## 1 Crossover point estimation

Assuming the cost of any single arithmetic operation is 1, we can compute the base cases and recurrence relation for Strassen's algorithm as follows:

$T(1) = 1$ since it only takes 1 multiplication of 2 integers in the case of $1 \times 1$ matrix.

Consider the case of $2 \times 2$ matrix:

- $P1, P2, P3, P4$ cost 2 each
- $P5, P6, P7$ cost 3 each
- $AE + BG$ and $CF + DH$ cost 3 each
- $AF + BH$ and $CE + DG$ cost 1 each

$T(2) = 4 \times 2 + 3 \times 3 + 2 \times 3 + 3 \times 1 = 25$

Consider the case of $4 \times 4$ matrix:

- $P1, P2, P3, P4$ each costs 4 to add/subtract two $2 \times 2$ submatrices and then $T(2)$ to multiply
- $P5, P6, P7$ each costs 8 to add/subtract two pairs of $2 \times 2$ submatrices and then $T(2)$ to multiply
- There are 8 operations to add/subtract $P_i$'s of size $2 \times 2$ to compute the four quadrants, and each costs 4

$T(4) = 4 \times \big(4 + T(2)\big) + 3 \times \big(2 \times 4 + T(2)\big) + 8 \times 4 = 247$

Assuming $n$ is a power of 2, we can generalize to the case of $n$ matrix:

$$\begin{aligned} T(n) &= 4\left(\left(\frac{n}{2}\right)^2 + T\left(\frac{n}{2}\right)\right) + 3\left(2\left(\frac{n}{2}\right)^2 + T\left(\frac{n}{2}\right)\right) + 8\left(\frac{n}{2}\right)^2 \\ &= 18\left(\frac{n}{2}\right)^2 + 7T\left(\frac{n}{2}\right) \\ &= n^{\log 7} \end{aligned}$$

Using the conventional algorithm, it takes $n$ multiplications and $n-1$ sums to compute each of the $n^2$ entries in the product matrix, for a total of $n^2(2n-1)$.

We need to find the value of $n$ where it costs the same amount using only the conventional algorithm or using Strassen's algorithm then switching to conventional. That is, we need to solve for:

$$\begin{aligned} n^2(2n-1) &= 18\Big(\frac{n}{2}\Big)^2 + 7\Big(\frac{n}{2}\Big)^2\Big(2\cdot\frac{n}{2}-1\Big) \\ 2n^3 - n^2 &= \frac{9}{2}n^2 + \frac{7}{4}n^3 - \frac{7}{4}n^2 \\ \frac{1}{4}n^3 - \frac{15}{4}n^2 &= 0 \\ n &= 15 \end{aligned}$$

Therefore, it is optimal to use the crossover point of 15.

# 2 Experimentation of variant Strassen's

## 2.1 Padding

To handle the cases when $n$ is odd, my algorithm first checks whether $n$ is a power of 2. If it's not, I calculate the next lowest power of 2 greater than $n$ and pad the matrix with zeros such that this power of 2 is the new size of the matrix. For example, when $n = 257$ and the next power of 2 is 512, I pad the matrices with zeros from columns 258 to 512 and rows 258 to 512, then proceed to multiply the two new $512 \times 512$ matrices using the variant Strassen's algorithm. Padding with zeros does not affect the multiplication result since each entry $c_{ij}$ where $i, j \leq 257$ of the new product matrix is still the same as if we were multiplying the two orginal matrices.

$$c_{ij} = \sum_{k=1}^{257} a_{ik}b_{kj} + \sum_{k=258}^{512} a_{ik}b_{kj} = \sum_{k=1}^{257} a_{ik}b_{kj} + 0 = \sum_{k=1}^{257} a_{ik}b_{kj}$$

Therefore, after multiplying the two $512 \times 512$ matrices, my algorithm returns the final product which is just the upper left submatrix of size $257 \times 257$ of the Strassen's result.

In order for Strassen's to work, we need to be able to keep dividing $n$ by 2. So whenever $n$ is odd, we need to pad the matrix with $2n+1$ zeros to get a new even value of $n$. Therefore, any matrix of size $n$ that is not a power of 2 will need a certain number of zeros added, and it does not matter in terms of correctness whether to pad all the zeros before calling Strassen's as I did or to pad zeros every time $n$ becomes odd. However, my algorithm is more efficient since it does not check whether $n$ is odd or even more than once.

## 2.2 Optimization

Coding in Python, I opted to represent $2D$ matrices using NumPy arrays rather than regular Python lists, since it is more efficient (and allowed) to use NumPy vectorized operations for matrix additions and subtractions than to loop through lists and add/subtract each pair of entries. However, in the conventional algorithm where we must loop through $i, j, k$ to multiply and add to get each entry $c_{ij}$ of the final product, I convert the Nympy matrices into 2D lists instead since it is more efficient to iterate through elements of a list than a Numpy array.

To further improve the conventional algorithm, I switched the order of the nested loops by looping through $k, i, j$ to compute $c_{ij} \mathrel{+}= a_{ik} \cdot b_{kj}$. That is, my algorithm goes through all elements of each row in both matrices before moving onto the next row, and repeats this process $n$ times. This is faster compared to the standard implementation $i, j, k$, which loops through the rows of one matrix and the columns of another.

I attempted to improve Strassen's by avoiding "unnecessary" memory allocation. Instead of creating new submatrices to store the 4 quadrants of each matrix and to store $P_i$'s, I store them directly on the two original matrices. For example, to compute $P1$, I first compute $F - H$ and store it on the $F$ (second) quadrant of matrix 2. Then I compute $P1 = A \cdot (F - H)$ by recursing Strassen's on the Numpy array slicing of the first quadrant of matrix 1 and second quadrant of matrix 2, and store it on a temporary matrix. After adding $P1$ to the second and fourth quadrants of the final product, I reset $F$ by adding $H$ back to $F$ to get the original matrix 2. I repeat this process for $P2 - P7$ and reuse the same temporary matrix the store them. That way, instead of allocating extra space to 15 matrices $A - F$ and $P1 - P7$, I only need 1 temporary matrix. However, in practice, I found the original implementation - simply creating new submatrices $A - H$ and $P1 - P7$ and proceed with the usual calculations- to be faster for large $n$. Although they both passed the autograder for correctness and runtime (for 60 points), the original implementation takes 1.00, 5.34, 38.80, and 279.15 seconds for $n$ =128, 256, 512, and 1024 respectively, while the space-saving implementation takes 1.05, 8.15, 52.97, and 378.30 seconds. I also found the difference to be the same when I tried different values of the crossover point.

To find the optimal crossover point, I generated random matrices with $n$ =128, 256, 512, and 1024 where each entry is randomly selected to be 0 or 1. For each value of $n$, I tested different values of the crossover point to find the smallest runtime. It does not matter whether the matrix size I use to test on is odd or even, since any matrix where $n$ is not a power of 2 would be padded with zeros using the padding strategy I described above. Thus, the runtime when $n = 257$, for example, would be roughly equal to the runtime when $n = 512$ plus a few extra seconds for padding. The optimal crossover point I found with my implementation is 4.

## 3 Triangles in random graphs

*Figure 1* is a plot of the number of triangles from generated random graphs against the expected number of triangles for different values of $p$, and *Table 1* shows the actual numbers. We can see that the numbers of triangles counted from the generated graphs are very close to the expected numbers.

Figure 1

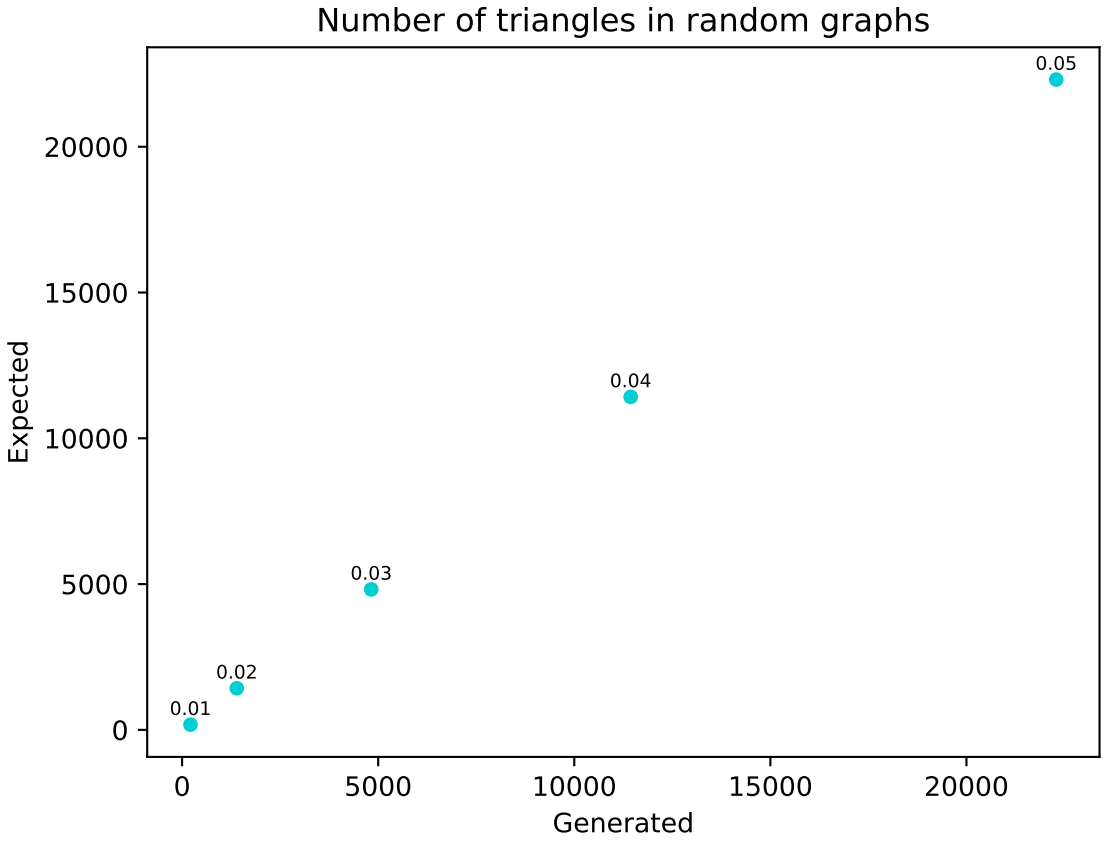


Table 1: Number of triangles in random undirected graphs with 1024 vertices

| $p$ | Expected | Generated |
|---|---|---|
| 0.01 | 178.43 | 214 |
| 0.02 | 1427.46 | 1395 |
| 0.03 | 4817.69 | 4823 |
| 0.04 | 11419.71 | 11438 |
| 0.05 | 22304.13 | 22289 |