CS 124 Programming Assignment 2: Spring 2022

Your name(s) (up to two): Burak Ufuktepe

Collaborators: None
No. of late days used on previous psets: 10
No. of late days used after including this pset: 12

Analytical Approach

Let $T_c(n)$ and $T_s(n)$ be the time required to multiply two $n \times n$ matrices using the conventional matrix multiplication algorithm and Strassen's algorithm, respectively.

Let $T_c(n)$ be the time required to multiply two $n \times n$ matrices using the conventional matrix multiplication algorithm. The conventional matrix multiplication requires n multiplications and $n-1$ additions for each element and there are n^2 elements in the matrix. Therefore, the runtime for the conventional matrix multiplication is:

$$T_c(n) = n^2(2n-1)$$

Let $T_s(n)$ be the time required to multiply two $n \times n$ matrices using Strassen's algorithm. Strassen's algorithm performs 7 multiplications and 18 additions/subtractions of matrices of size $\lceil n/2 \rceil$. The reason why we use a ceiling function is because n needs to be divided in half evenly, if n is odd we would need to pad the matrix with one extra zero row and one extra zero column. Thus, we can express the recurrence relation as follows:

$$T_s(n) = 7T_s(\lceil n/2 \rceil) + 18(\lceil n/2 \rceil)^2$$

To find the cross-over point we need to find the value for n when $T_s(n) \geq T_c(n)$. So, we can simply use the equality $T_s(n) = T_c(n)$ and solve for n.

$$n^2(2n-1) = 7T_s(\lceil n/2 \rceil) + 18(\lceil n/2 \rceil)^2$$

$$n^2(2n-1) = 7\lceil n/2 \rceil^2(2\lceil n/2 \rceil - 1) + 18(\lceil n/2 \rceil)^2$$

If n is even, we have:

$$n^2(2n-1) = 7(n/2)^2(2(n/2)-1) + 18(n/2)^2$$

$$n = 15$$

This shows when n is even and less than 15, switching to the conventional algorithm will be more optimal than using Strassen's algorithm.

If n is odd, we will have to pad the matrices with one extra zero row and one extra zero column. Let $n = 2m + 1$. After padding the matrices and dividing by half we obtain:

$$\lceil n/2 \rceil = \frac{2m + 2}{2} = m + 1$$

So we have:

$$(2m + 1)^2(2(2m + 1) - 1) = 7(m + 1)^2(2(m + 1) - 1) + 18(m + 1)^2$$

$$m \approx 18.1$$

$$n \approx 37.2$$

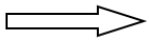
This shows when n is odd and less than 37.2, switching to the conventional algorithm will be more optimal than using Strassen's algorithm.

Implementation

Data Layout Optimization

Splitting matrices takes up a significant portion of the actual runtime in Strassen's algorithm. To speed up this process, instead of using a standard row-major ordering, Morton ordering is used to represent $n \times n$ matrices. Morton ordering takes a 2D array stored in row-major order and arranges the matrix in $m \times m$ block arrays where m is the size of the "base case". This is illustrated in Figure 1 for $m = 2$ and Figure 2 for $m = 3$.

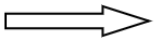
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63



0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Figure 1: Row-Major Ordering vs Morton Ordering (2 x 2 blocks)

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35



0	1	2	9	10	11
3	4	5	12	13	14
6	7	8	15	16	17
18	19	20	27	28	29
21	22	23	30	31	32
24	25	26	33	34	35

Figure 2: Row-Major Ordering vs Morton Ordering (3 x 3 blocks)

This way, we end up with a 1D array and each quadrant is stored contiguously in memory as shown in Figure 3. To partition an $n \times n$ matrix into four $n/2 \times n/2$ matrices we can simply slice the 1D array into four parts without having to iterate over each element.

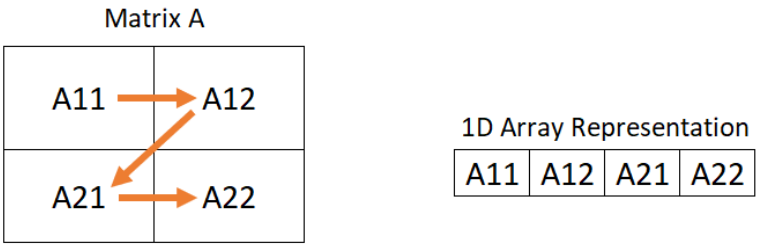


Figure 3: 1D Array Representation of a Matrix using Morton Ordering

To compare the performance of Morton Ordering to Row-Major Ordering, several trials are conducted using random matrices where each entry is randomly selected to be 0, 1 or 2. Matrices of sizes 1536×1536, 1792×1792 and 2048×2048 are tested using various cross-over points. Test results are given in Table 1. Furthermore, for each matrix size, runtime vs cross-over point plots are shown in Figure 4, Figure 5, and Figure 6 where the minimum runtimes are highlighted with red dots.

Table 1: Morton-Ordering vs Row-Major Ordering Runtime Comparison

n	Cross-Over Point	Runtime (s)	
		Morton-Ordering	Row-Major Ordering
1536	6	243	337
1536	12	187	236
1536	24	176	202
1536	48	181	196
1536	96	192	202
1792	7	347	467
1792	14	278	343
1792	28	273	308
1792	56	277	303
1792	112	288	317
2048	4	704	1036
2048	8	475	624
2048	16	393	470
2048	32	398	422
2048	64	414	445
2048	128	451	468

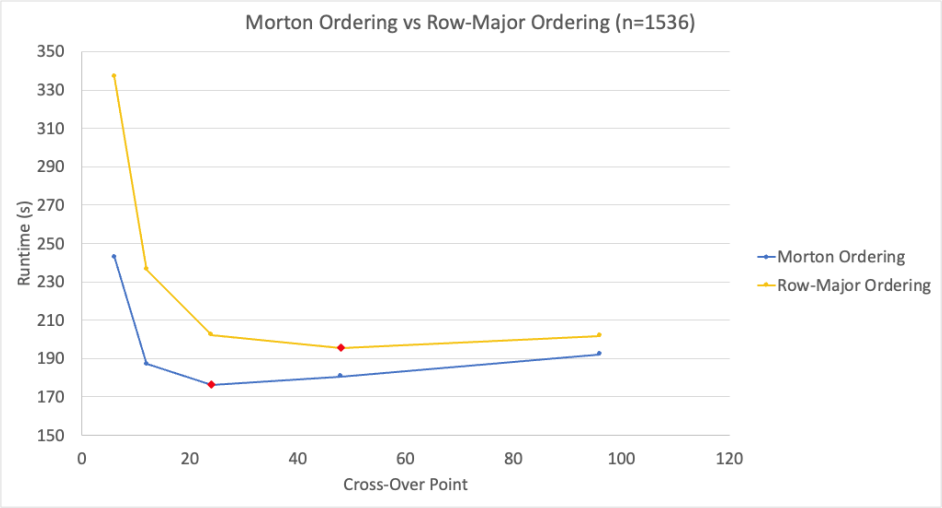


Figure 4: Morton Ordering vs Row-Major Ordering (n=1536)

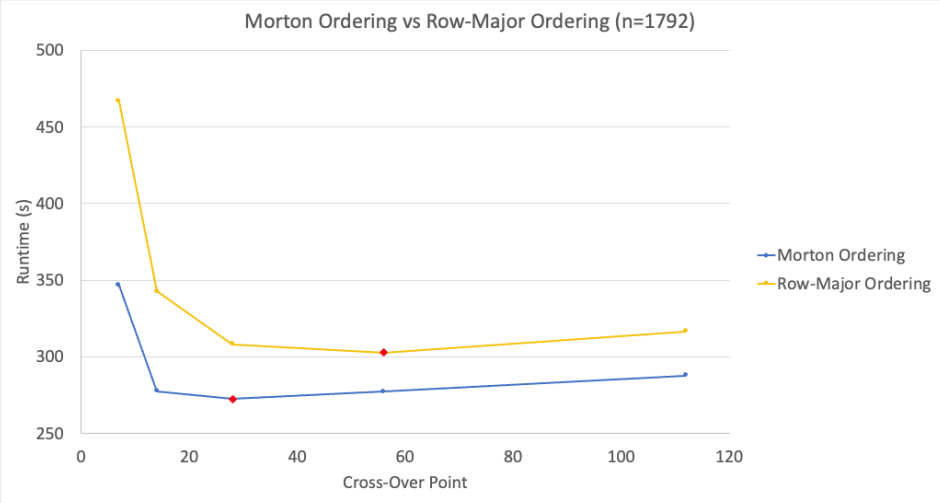


Figure 5: Morton Ordering vs Row-Major Ordering (n=1792)

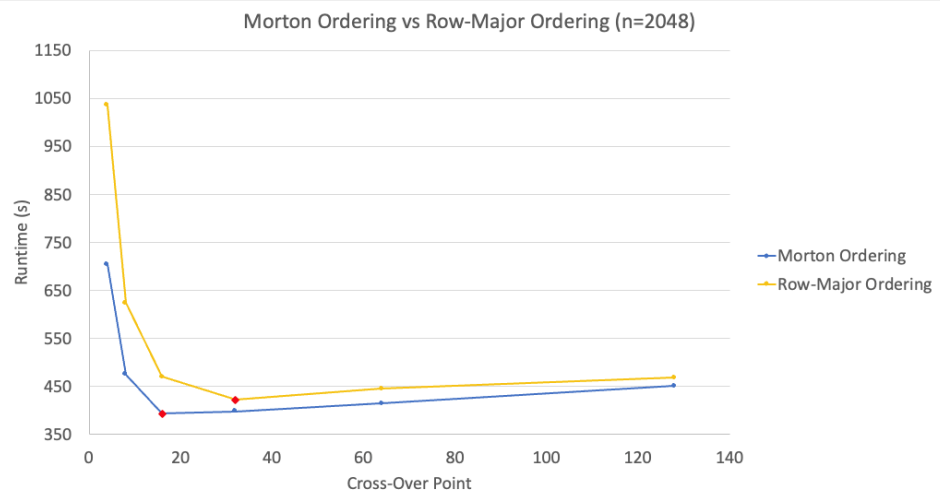


Figure 6: Morton Ordering vs Row-Major Ordering (n=2048)

Morton Ordering shows a noticeable performance gain over Row-Major Ordering (up to 32%). The largest performance gain is observed when the cross-over point is small. This is because using a smaller cross-over point results in more recursive calls in Strassen's algorithm, as a result the number of matrix partition operations increases. Since splitting up a matrix in Morton Ordering is a lot cheaper compared to Row-Major Ordering, Morton Ordering yields better results in terms of runtime.

Also note that the optimum cross-over point for Morton Ordering is lower than the optimum cross-over point for Row-Major Ordering. This is because Strassen's algorithm runs more efficiently when the matrices are laid out in Morton Ordering. Whereas, when we use Row-Major ordering, the algorithm spends too much time tying to split up matrices.

Conventional Matrix Multiplication Optimization

In order to multiply two matrices that are Morton-ordered, the following procedure is used where a and b are matrices to be multiplied, c is the resultant Morton-ordered matrix that is initially filled with zeros, m is the size of the matrix, and idx is the current index of c that will be populated. Note that this function is part of a recursive function, therefore we need an index argument (idx) to identify the starting index for c.

As you can see in the for loop of lines 3-8, the procedure accesses $a[i * m + k]$ multiple times and at each time, it performs a computation to find the relevant index of a. Furthermore, due to Morton ordering, the procedure jumps between contiguous blocks of memory which slows down the process.

Algorithm 1 Non-optimized Conventional Matrix Multiplication

```
1: procedure NON-OPTIMIZED-MATRIX-MULT(a, b, c, m, idx)
2:     for i = 1 to m do
3:         for j = 1 to m do
4:             r = 0
5:             for k = 1 to m do
6:                 r = r + a[i * m + k] * b[j + k * m]
7:             c[idx] = c[idx] + r
8:             idx = idx + 1
```

When we bring a block of data into the cache, we would like it to contain as much useful data as possible and perform as much useful work as possible on it before removing it from the cache. Therefore, to optimize this procedure, the frequently accessed elements of a are stored in a temporary array as shown in line 3 of the following procedure. Then, in line 7, this array is used to access the relevant element of a. Since $temp$ is a much smaller array it provides better cache performance (due to index locality). Furthermore, there is no computation involved to find the relevant index hence the runtime is further reduced.

Algorithm 2 Optimized Conventional Matrix Multiplication

```
1: procedure OPTIMIZED-MATRIX-MULT(a, b, c, m, idx)
2:     for i = 1 to m do
3:         temp = [a[i * m + 1], a[i * m + 2], ··· , a[i * m + m]]
4:         for j = 1 to m do
5:             r = 0
6:             for k = 1 to m do
7:                 r = r + temp[k] * b[j + k * m]
8:             c[idx] = c[idx] + r
9:             idx = idx + 1
```

Several trials have been conducted to compare the runtimes of these two procedures. Random matrices of sizes from 500×500 up to 2000×2000 are used and the results are given in Figure 7. It is observed that the optimized procedure runs 21% to 24% faster than the non-optimized procedure.

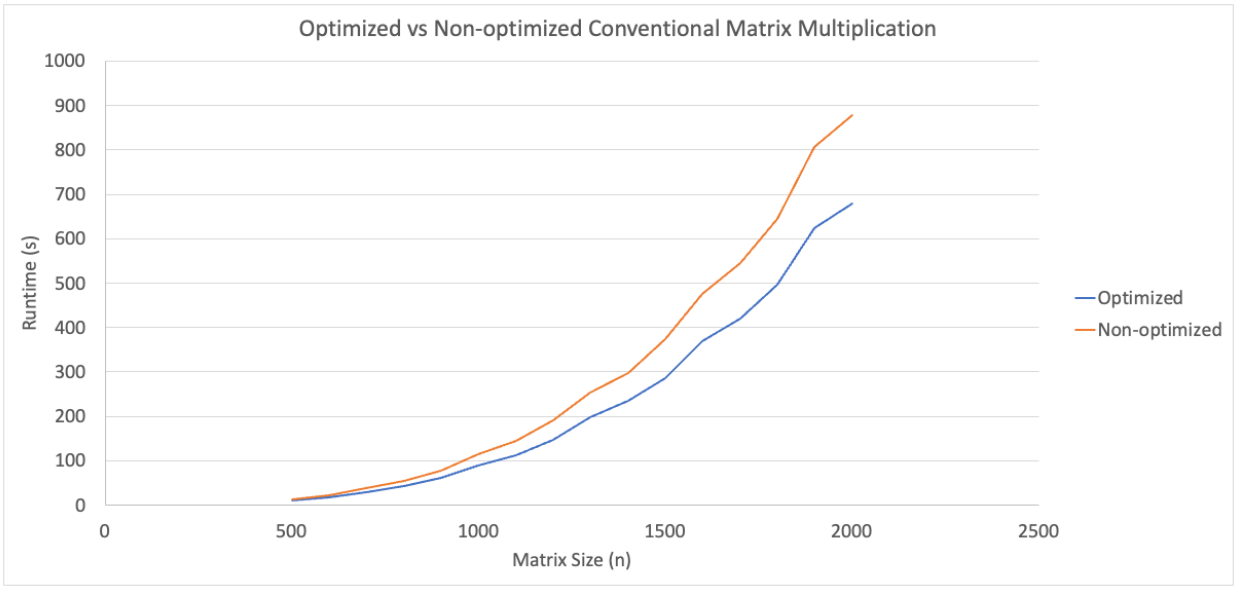


Figure 7: Optimized vs Non-optimized Conventional Matrix Multiplication

Padding

Strassen's algorithm recursively divides $n \times n$ matrices into four $n/2 \times n/2$ matrices. That means at each recursive call n must be divisible by 2. The obvious method to resolve this issue would be to pad the original matrix to the next power of 2 with zero rows and zero columns. However, this would be an expensive approach because we would have to almost double the size of the original matrix if its original size is just over a power of 2 (i.e $n = 1025$).

Instead, we use the following approach: we first find the size of the "base case" for our Morton-ordered array. Then we keep doubling that size until we reach or exceed the size of our original matrix to find the minimum required size that we need to perform Strassen's algorithm. That is:

1. Let $p = n$. Repeatedly divide p in half, each time taking the ceiling, until p is less than or equal to the cross-over point. This will be equal to the size of the "base case" for our Morton-ordered array.
2. Then repeatedly double p until $p \geq n$.
3. Pad the original matrix with zero rows and zero columns until we obtain a matrix of size $p \times p$.

In other words, we are padding the matrix just enough so that Strassen's algorithm can reach the "base case". One of the main advantages of this approach is, since the padding is done upfront, we do not have to worry about odd-sized matrices while performing Strassen's algorithm.

For example, assume $n = 1025$ and the cross-over point is 37. We find the size of the base case as follows:

$$\lceil 1025/2 \rceil = 513 \rightarrow \lceil 513/2 \rceil = 257 \rightarrow \lceil 257/2 \rceil = 129 \rightarrow \lceil 129/2 \rceil = 65 \rightarrow \lceil 65/2 \rceil = 33$$

Then, to find the minimum required matrix size, we repeatedly double the base case until we reach or exceed n:

$33 \times 2 = 66 \rightarrow 66 \times 2 = 132 \rightarrow 132 \times 2 = 264 \rightarrow 264 \times 2 = 528 \rightarrow 528 \times 2 = 1056$

Finally, we pad our original 1025×1025 matrix until we reach a size of 1056×1056.

Mulltiprocessing

To reduce the overall runtime, Python's multiprocessing module is utilized while performing the experimental analysis to find the optimum cross-over point. The analysis is run on a laptop with a 10-core Apple M1 Pro CPU. For the initial trials, 8 parallel processes were used. However, due to overloading the CPU a high variance was observed in runtimes. Therefore, the experimental analysis was conducted using 4 parallel processes.

Experimental Analysis

An experimental analysis is performed to find the optimum cross-over point. As explained in the previous section, our implementation pads the given matrices (if necessary) before performing Strassen's algorithm. Therefore, the size of the matrices is always converted from n to $q \cdot 2^k$ where q is the size of the "base case" such that $q \leq n_0$ and k is the smallest integer that satisfies $n \leq q \cdot 2^k$. In other words, it doesn't matter if we multiply odd-sized matrices or even-sized matrices because our algorithm always converts the given matrix to an even-sized matrix by padding the matrix before executing Strassen's algorithm.

Moreover, it is observed that this conversion process (padding) takes only a fraction of the total runtime. Hence, to simplify the analysis, padding is avoided by selecting matrices of sizes $n = n_0 \cdot 2^k$ where n_0 is the cross-over point.

Based on our analytical analysis, the actual optimum cross-over point is anticipated to be in the range of 10-30. However, to better understand the behavior of our implementation, cross-over points at various increments from 4 to 160 are considered as shown below.

Cross-Over Points (n_0): 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 20, 22, 24, 26, 28, 30, 32, 36, 40, 44, 48, 52, 56, 60, 64, 72, 80, 88, 96, 104, 112, 120, 128, 144, 160.

To test these cross-over points, the following 16 matrix dimensions are used.

Matrix Dimensions (n): 768, 832, 896, 960, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, 2048, 2304, 2560, 2816.

For each dimension, different cross-over points are considered. An example is presented in Table 2 for $n = 1536$.

Table 2: Cross-Over Points Considered for n=1536

$n = n_0 \cdot 2^k$	n_0
$1536 = 6 \cdot 2^8$	6
$1536 = 12 \cdot 2^7$	12
$1536 = 24 \cdot 2^6$	24
$1536 = 48 \cdot 2^5$	48
$1536 = 96 \cdot 2^4$	96

The runtime vs cross-over point graphs are shown in Figure 8 and Figure 9 for $n < 1500$ and $n > 1500$, respectively. In these graphs, the minimum runtime for each matrix size is highlighted with a red dot. The results are also tabulated in the Appendix section (see Table 5) and the optimum cross-over point for each matrix size is given in Table 3.

Table 3: Optimum Cross-Over Points

n	Optimum Cross-Over Point
768	24
832	26
896	28
960	30
1024	16
1152	18
1280	20
1408	22

n	Optimum Cross-Over Point
1536	24
1664	26
1792	28
1920	30
2048	16
2304	18
2560	20
2816	22

Note that the size of the matrices in Figure 8 are doubled in Figure 9. In both graphs we see the same trends. That is, the optimum cross-over point for a matrix of size $n \times n$ and $2n \times 2n$ are the same. Hence, we would expect to obtain the same optimum cross-over points if we were to double the matrices even further.

In both graphs we observe that, for every matrix size, the minimum cross-over point that is larger than 15 yields the minimum runtime. More specifically, as shown in Table 3, the optimal cross-over point varies in the range of 16 to 30. Hence, based on the experimental results, we can define the optimal cross-over point as 30 which is close to our analytical cross-over point of 15 (for even-sized matrices).

Also, we observe that for $n = 768, 832, 896, 960, 1536, 1664, 1792,$ and 1920 the runtime difference between the optimum cross-over point and the next smaller cross-over point is negligible. Therefore, the experimental cross-over point may further be decreased by optimizing the memory usage of Strassen's algorithm.

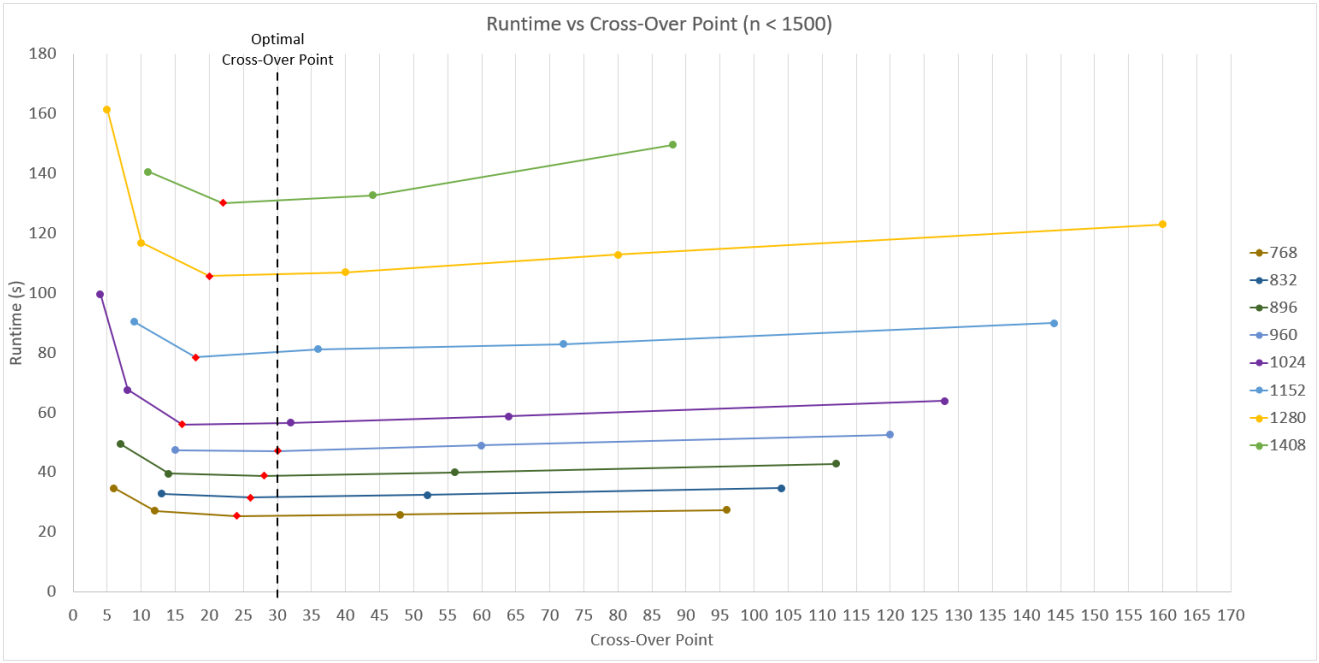


Figure 8: Runtime vs Cross-Over Point (n < 1500)

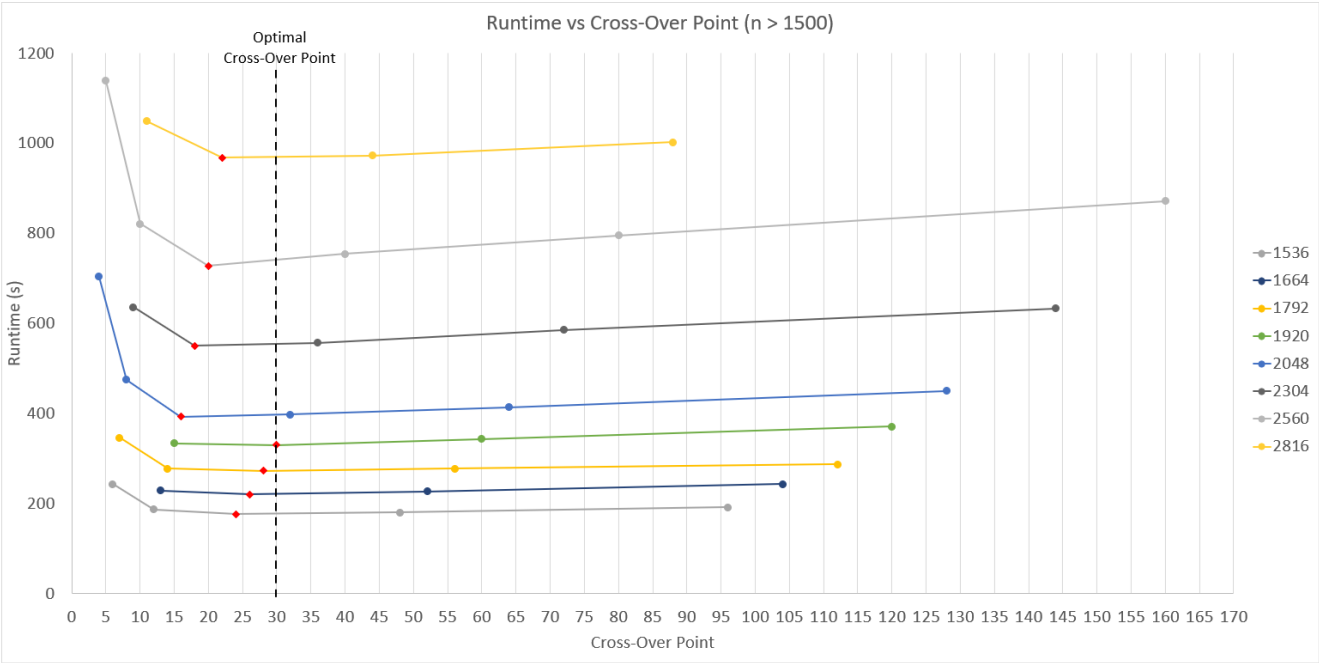


Figure 9: Runtime vs Cross-Over Point (n > 1500)

Triangle in Random Graphs

Random graphs on 1024 vertices are generated where in each graph edges are included with probabilities $p = 0.01, 0.02, 0.03, 0.04$, and 0.05. Strassen's algorithm is used to count the number of triangles in each of these graphs, and the results are compared to the expected number of triangles, which is $\binom{1024}{3}p^3$. For each probability, 10 graphs are generated, and the average number of triangles are calculated. The results are given in Table 4.

Table 4: Expected vs Actual Number of Triangles

p	Expected Number of Triangles in Graph	Avg. Number of Triangles in Graph
0.01	178	179
0.02	1427	1454
0.03	4818	4825
0.04	11420	11515
0.05	22304	22327

We see that the maximum difference between the expected and average number of triangles is less than 2%. The maximum difference may further be reduced by perfoming more trials.

Appendix

Table 5: Experimental Results

n	Cross-Over Point	Runtime (s)
768	6	35
768	12	27
768	24	25
768	48	26
768	96	27
832	13	33
832	26	31
832	52	32
832	104	35
896	7	49
896	14	40
896	28	39
896	56	40
896	112	43
960	15	47.4
960	30	47.1
960	60	49
960	120	53
1024	4	100
1024	8	68
1024	16	56.0
1024	32	56.5
1024	64	59
1024	128	64
1152	9	90
1152	18	78
1152	36	81
1152	72	83
1152	144	90
1280	5	161
1280	10	117
1280	20	106
1280	40	107
1280	80	113
1280	160	123
1408	11	141
1408	22	130
1408	44	133
1408	88	150

n	Cross-Over Point	Runtime (s)
1536	6	243
1536	12	187
1536	24	176
1536	48	181
1536	96	192
1664	13	229
1664	26	221
1664	52	228
1664	104	244
1792	7	347
1792	14	278
1792	28	273
1792	56	277
1792	112	288
1920	15	334
1920	30	331
1920	60	344
1920	120	371
2048	4	704
2048	8	475
2048	16	393
2048	32	398
2048	64	414
2048	128	451
2304	9	636
2304	18	551
2304	36	557
2304	72	586
2304	144	633
2560	5	1139
2560	10	822
2560	20	728
2560	40	754
2560	80	795
2560	160	872
2816	11	1049
2816	22	969
2816	44	973
2816	88	1002