Optimized Algorithm Analysis for Odd Dimensions Magic Squares

Submission Date · June 2018

Author **:** Joseph Assaker - 53190
~ 3rd Year Computer Science Student

Supervisor **:** Pr. Fadi Dabaghi

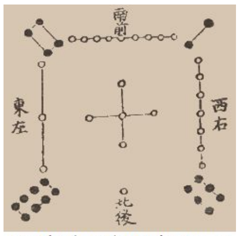
Optimized Algorithm Analysis for Odd Dimensions Magic Squares

Abstract *- The mathematical study of magic squares typically deals with its construction, classification, and enumeration. Although completely general methods for producing all the magic squares of all orders do not exist; historically, there is three main general techniques for producing magic squares: by bordering method, by making composite magic squares, and by adding two preliminary squares. In the present work, and in addition to these mentioned techniques, we use more specifically the strategy based on the continuous enumeration method that reproduces specific patterns. Hence, we'll study and analyze this strategy by elaborating and developing optimized efficient algorithms, taking into account the recursive uses aspect of the creation of a magic square.*

Keywords: ***Magic Squares, Perfect Magic, Space Filling, Symmetry, Swap.***

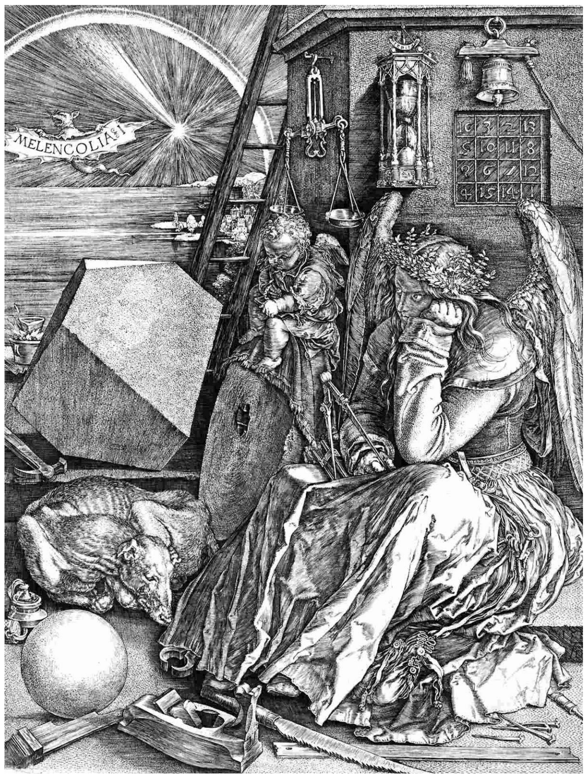
I. INTRODUCTION

Magic squares have been the subject of entertainment and interest to many mathematicians and math-enthusiasts alike for hundreds of years. Let us consider a square matrix (n x n) whose entries are distinctly the integers (1, . . . , n^2). A normal magic square is such that the elements sum of any row, column, or major and minor traces (diagonals) is equal to one constant μ [WM001]. Magic squares of order 3 and 4 have appeared in paintings, literature, and artifacts dated as far back as 650 BC. One of the first magic squares found in ancient Chinese literature is the Lo Shu square, which is told to have been painted on the shell of a sea turtle [WK001].



Lo-Shu (3 x 3) Magic Square

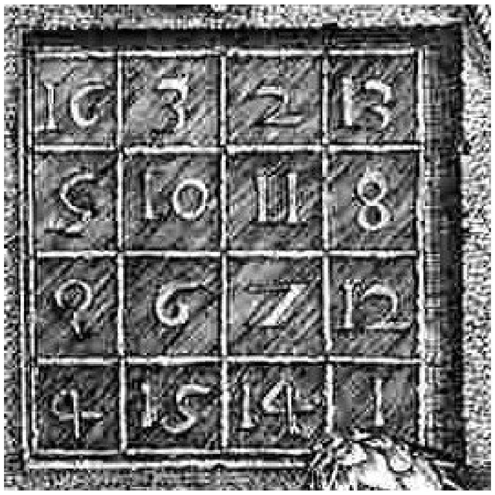
One of the most notable magic squares is the order 4 magic square in Albrecht Durer's engraving Melencolia I [WK002].



Melencolia I - "You can find in the upper right corner the famous (4 x 4) magic square".

Indeed, the Durer magic square has many fascinating properties. Aside the rows and columns (as well as major/minor diagonal) elements' summing to the magic number 34, the sum can also be found in each 2 by 2 quadrant, the center 2 by 2 square, the four outer corner squares, and the four outer corner squares of each 3 by 3 subsquare.

Furthermore, the sum can be found clockwise from the corners (3 + 8 + 14 + 9) and counter-clockwise (5 + 15 + 12 + 2). What's most fascinating about the Durer square is the bottom row: 1514 was the year the painting was made, and 4 and 1 correspond to the letters D and A, the initials of Albrecht Durer.



Melencolia I - "a close-up on the magic square".

Nowadays, magic squares applications are widely utilized in the world of computation, ranging from IP Addressing to Image Encryption [AR001] [AR002] and Photography Matrix Metering, as well as numerous physical benefits and properties for the magic squares as: Mass center, Physical moment of inertia, Dipole moment, Water retention on mathematical surfaces, etc… [AR003]

In paragraph II, we will pose the problem and introduce the different definitions of a magic square, as well as introducing the space filling algorithms, consequently explaining some useful elementary functions that will be used later on in our algorithms.

In the third paragraph, we give the numerical method and the implementation process of our algorithms, along with a detailed analysis, followed by a comparison between them at the theoretical level.

In the results dedicated paragraph, many test cases are performed, comparing the different suggested algorithm as well as their efficiency, confirming the analysis developed in the third paragraph.

Finally, some concluding remarks are given, along with various recommendations, based on the user's hardware specifications, and some guidelines for the follow up of this work, in particular parallel algorithmic analysis .

II. PROBLEM STATEMENT

This work seeks to unveil the following question: How to develop algorithms that creates magic squares while optimizing that process.

For any normal magic square of order n, one can show that:

$$\mu = n^2 * (n^2 + 1) / 2n = n * (n^2 + 1) / 2$$

which, for the case of n = 5, is the number 65. In this study, only odd dimensions squares are considered (as the dimension 1 is a trivial perfect magic square, the dimensions n given for the squares in this work will be $n \geq 3$ and for our illustrations we will show the example of n = 5 for better visibility and clarity).

Given the diverse uses of magic squares and the need to provide a magic square or a sequence of magic squares reliably and quickly, the generation process of magic squares needs to be optimized. In this work, three types of squares are defined: (a) Average Magic Square (b) Normal Magic Square (c) Perfect Magic Square.

(a) Average Magic Square: It is a square where the sums of each column, row and major/minor diagonals along with the pseudo-diagonals will have an average of μ. Thus the sums will be symmetrically around the magic constant μ.

14	20	21	2	8	≈ 65
4	10	11	17	23	≈ 65
19	25	1	7	13	≈ 65
9	15	16	22	3	≈ 65
24	5	6	12	18	≈ 65
≈ 65	≈ 65	≈ 65	≈ 65	≈ 65	≈ 65

(5 x 5) Average magic square example

23	6	19	2	15	= 65
4	12	25	8	16	= 65
10	18	1	14	22	= 65
11	24	7	20	3	= 65
17	5	13	21	9	= 65
= 65	= 65	= 65	= 65	= 65	= 65

(5 x 5) Perfect magic square example

(b) Normal Magic Square: It is a square where the sums of each column and row is the magic constant μ. Furthermore the sums of the major/minor diagonals along with the pseudo-diagonals will have an average of μ.

22	3	9	15	16	= 65
8	14	20	21	2	= 65
19	25	1	7	13	= 65
5	6	12	18	24	= 65
11	17	23	4	10	= 65
= 65	= 65	= 65	= 65	= 65	≈ 65

(5 x 5) Normal magic square example

(c) Perfect Magic Square: It is a square where the sums of each column, row and major/minor diagonals along with the pseudo-diagonals are equal to the magic constant μ.

Given a space filling pattern, being the Chess' Knight step (or the L shape step, which is basically going two steps in one direction and one step in another perpendicular direction), one can aim to create a magic square of any type by starting at any position with the value 1 given to it and continuing the pattern incrementally till the affection of n^2 elements. After finding specific ways to create perfect magic squares, the big problem lies in specifying more optimized ways to create more perfect magic squares from the already created perfect magic squares without repeating the complexity of a specific space filling algorithm being $O(n^2)$.

All of the below algorithms will require the use of the following basic functions:

i) moveU (Move Up): Which will increase the row numbering by 1. When the value exceeds the maxing row numbering (N-1), the returned value is 0.

ii) moveD (Move Down): Which will decrease the row numbering by 1. When the value is inferior to the minimum row numbering (0), the returned value is N-1.

iii) moveR (Move Right): Which will increase the column numbering by 1. When the value exceeds the maxing column numbering (N-1), the returned value is 0.

iv) moveL (Move Left): Which will decrease the column numbering by 1. When the value is inferior to the minimum column numbering (0), the returned value is N-1.
v) normalize: Which will take any out of range value (range being [0 : N-1]) and normalize it to fit the interval.

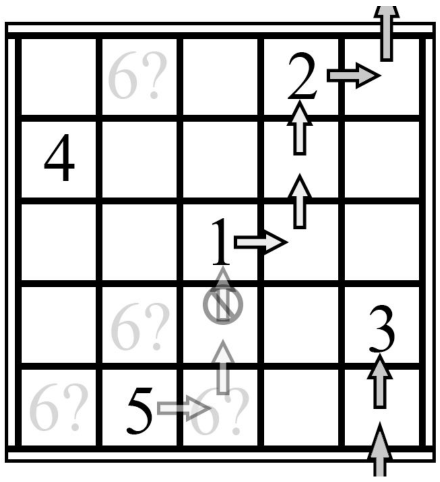
III. METHODS AND ALGORITHMS

A. Space Filling Algorithms

As discussed earlier, the L shape step pattern will be elaborated to check if any derivation of this pattern will lead to any useful results. Specifically, four variants of the L shape step will be used in this study: (1) Right-Right-Up (2) Right-Up-Up (3) Left-Left-Down (4) Left-Down-Down.

Working with the L shape step pattern, one can find that the pattern is stuck after N position fillings. As it is going one step in a given direction, at each iteration of the pattern; thus filling one element in each of the N lines and columns of the square after N position fillings, and returning to the initial starting position at the N+1 iteration. Solving this can be achieved in several ways, for example moving Up, Down, Right or Left when the pattern is stuck, and continuing the normal pattern until we get stuck again. And as demonstrated earlier, the pattern will be stuck at every N iterations so we will have to “unblock” the pattern with any given decision at every k*N iterations (k ≥ 1).

With careful testing, one can find that by making the decision of unblocking the pattern with the step opposite of that the pattern weights the most (does 2 times, ex: Right-Right-Up, Right is the most weighted step, it’s opposite would be Left) a perfect magic square will be created with any starting position.



(5 x 5) Square: Right-Right-Up pattern stuck at the N+1 iteration.

The four patterns defined earlier will be modified to satisfy the current goal to become:

(1) Right-Right-Up|Left.
(2) Right-Up-Up|Down.
(3) Left-Left-Down|Right.
(4) Left-Down-Down|Up.

Being now at the disposal of four algorithms generating each a perfect magic square at any given start position, we now can create 4*N^2 perfect magic squares for any odd dimensions squares; which is a total of 100 for n = 5, 40804 for n = 101 and so on.

(1) Right-Right-Up|Left algorithm in C++:

```
void RightRightUp(int** M,int i,int j){
        int k,kk,c=1;
        M[i][j]=c;
        c++;
        for(k=0;k<N;k++){
                for(kk=0;kk<N-1;kk++){
                        j=moveR(j);
                        j=moveR(j);
                        i=moveU(i);
                        M[i][j]=c++;
                }
                j=moveL(j);
                M[i][j]=c++;
        }
        M[i][j]=1;
}
```

**(2), (3) and (4) algorithms are equivalent with different steps taken at each step.*

Having achieved that, there is still one exception for those algorithms, being multiples of 3, for which none of the unblocking decisions led to a perfect magic square. However the creation of a normal magic square can be achieved by making the decision of unblocking the pattern with the step that the pattern weights the most.
Four patterns will be defined as variations of the initial patterns to satisfy the current goal to become:
(1’) Right-Right-Up|Right.
(2’) Right-Up-Up|Up.
(3’) Left-Left-Down|Left.
(4’) Left-Down-Down|Down.
**(1’), (2’), (3’) and (4’) algorithms are equivalent to the (1) algorithm with different steps taken at each iteration.*

Note that every other variation of the pattern by making other unblocking decisions will lead to an Average Magic Square.

Observing furthermore those algorithms, one can show that different results of a single algorithm coming from different starting positions are actually matching with the square moving according to the starting position. So defining two algorithms: (5) NextRight (6) NextUp, that will translate the whole square right or up with the out of bound column/line moving to the opposite side of the square.

**NextLeft and NextDown are equivalent.*

(5) NextRight algorithm in C++:
```
void NextRight(int** M, int* J){
        int i,j,temp;
        for(i=0;i<N;i++)
                for(j=N-1;j>0;j--){
                        temp=M[i][j];
                        M[i][j]=M[i][j-1];
                        M[i][j-1]=temp;
                }
        (*J)=moveR(*J);
}
```

(6) NextUp algorithm in C++:
```
void NextUp(int** M, int* I){
        int i,j,temp;
        for(j=0;j<N;j++)
                for(i=N-1;i>0;i--){
                        temp=M[i][j];
                        M[i][j]=M[i-1][j];
                        M[i-1][j]=temp;
                }
        (*I)=moveU(*I);
}
```

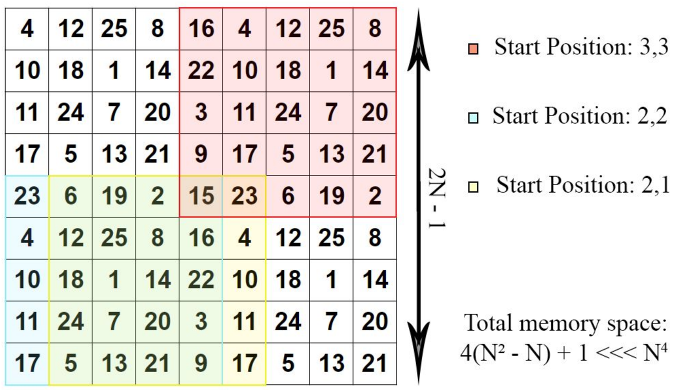
These algorithms, although having a complexity of O(n^2), but in real life testing they turn out to be much more efficient than reproducing another square via the patterns algorithms. This is mainly due to the absence of the numerous calls on moveU, moveD, moveL and moveR, which each needs to call its own function, run a test on the value then return it. But most crucially, those real life results unveil a critical efficiency criterion that is invisible in the complexity study of an algorithm, thus generally forgotten, and it is the time needed to call a certain variable in the considered square. In the patterns algorithms, we can see that we jump from value to value in a somewhat clean order until it gets to an edge where it has to now call a variable so far away from the current variable (ex: going from M[i][N-1] to M[i+2][0]). However in the translations algorithms, the values are only being interchanged between two contiguous variables (ex: between M[i][j] and M[i-1][j]), thus the call for another variable is relatively quick as it is right next to it. Furthermore tests will show that NextRight is around x3 more efficient, for small dimensions, then NextUp which clarify more about how the program’s matrix is stored (the access of the next column is much faster than the access of the next row).

Having achieved so much work to optimize the process of the generation of matching magic squares, one may request an instant access to all of the n^2 compositions produced by a single pattern. This can be achieved by a brute way of requesting $n^2 * n^2 = n^4$ memory space to store all of the squares. However a better solution would be to utilize the discussed property, that the squares are matching, and a square of $(2n - 1)^2$ can be stored with instant access to any square by changing the reference indexes. To accomplish this goal an algorithm (7) createAll_RightUpUp is developed.

**createAll_RightRightUp, createAll_LeftLeftDown, and createAll_LeftDownDown are equivalent.*

(7) createAll_RightUpUp algorithm in C++:

```
void createAll_RightUpUp(int** MM){
        int i,j;
        MM=(int **)calloc(NN+1,sizeof(int *));
        for(int y=0;y<NN;y++)
                MM[y] = (int *)calloc(NN,sizeof(int));
        MM[NN]=NULL;
        RightUpUp(MM,N/2,N/2);
        for(i=0;i<N;i++)
                for(j=N;j<NN;j++)
MM[i][j]=MM[i][j-N];
        for(i=N;i<NN;i++)
                for(j=0;j<N;j++)
MM[i][j]=MM[i-N][j];
        for(i=N;i<NN;i++)
                for(j=N;j<NN;j++)
MM[i][j]=MM[i-N][j-N];
}
```



4	12	25	8	16	4	12	25	8
10	18	1	14	22	10	18	1	14
11	24	7	20	3	11	24	7	20
17	5	13	21	9	17	5	13	21
23	6	19	2	15	23	6	19	2
4	12	25	8	16	4	12	25	8
10	18	1	14	22	10	18	1	14
11	24	7	20	3	11	24	7	20
17	5	13	21	9	17	5	13	21

Create all example for n = 5.

B. Mapping Between Space Filling Algorithms

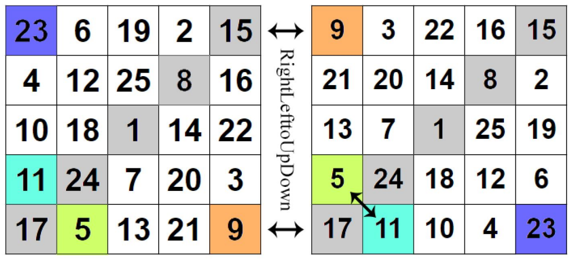
Having done this work, there is yet some final complex and crucial steps that needs to be optimized, and that is the mapping between the patterns algorithms. Having for example the RightRightUp|Left algorithm already created a perfect magic square at a given start position, it is widely possible to now request another perfect magic square with the same start position. To solve this issue, we need to develop algorithms capable of transforming a perfect magic square into another one while keeping the same starting position without calling another pattern algorithm with its low efficiency.

In this process four final algorithms are developed: (8) RightLefttoUpDown (9) UptoDown (9') UptoDown_2 (10) RightUptoLeftDown.

- *RightLefttoUpDown*

This algorithm aims to transform a pattern that weight more Right or Left, to the pattern that weight more Up or Down and vice versa. So it will transform RightUpUp to RightRightUp, LeftLeftDown to LeftDownDown, etc…

With a detailed analysis of the process, by choosing our starting position to be the center of the square, one can observe that those patterns are symmetric with respect to the major diagonal (swapping M[i][j] with M[j][i]).

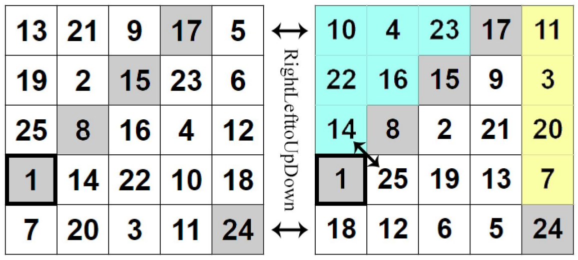


23	6	19	2	15
4	12	25	8	16
10	18	1	14	22
11	24	7	20	3
17	5	13	21	9

9	3	22	16	15
21	20	14	8	2
13	7	1	25	19
5	24	18	12	6
17	11	10	4	23

From Right-Up-Up to Right-Right-Up and vice versa (starting position: (2, 2)).

However things get trickier when the starting position moves. Indeed, as long as we keep the starting position in the major diagonal, the same symmetry is found. If not, moving the starting position out of the major diagonal, one can find that the square will be symmetric with respect to the pseudo-diagonal that the starting position belongs to. Having found that the symmetry could be any pseudo-diagonal that the starting position belongs to, a fixed set of positions need to be defined. These positions would be the intersections of the pseudo-diagonals with the column: j = 0 (in the case of having the starting position included in the major diagonal, this intersection would be the position (0,0)).



13	21	9	17	5
19	2	15	23	6
25	8	16	4	12
1	14	22	10	18
7	20	3	11	24

10	4	23	17	11
22	16	15	9	3
14	8	2	21	20
1	25	19	13	7
18	12	6	5	24

From Right-Up-Up to Right-Right-Up and vice versa (starting position: (2, 2)).

In the above case, the position we need to find is (1,0). After finding that position, the algorithm is split into two phases: Phase 1 consists of swapping the elements that form a square of (n-I × n-I) (I being the row of the fixed position) starting from the fixed position (swapping M[i][j] with M[j+I][i-I]). Phase 2 consists of swapping the remaining elements which are still symmetric with respect to the pseudo-diagonal and requires special steps to find their "virtual" symmetric and matching it with their actual symmetric in the square.

(8) RightLefttoUpDown algorithm in C++:

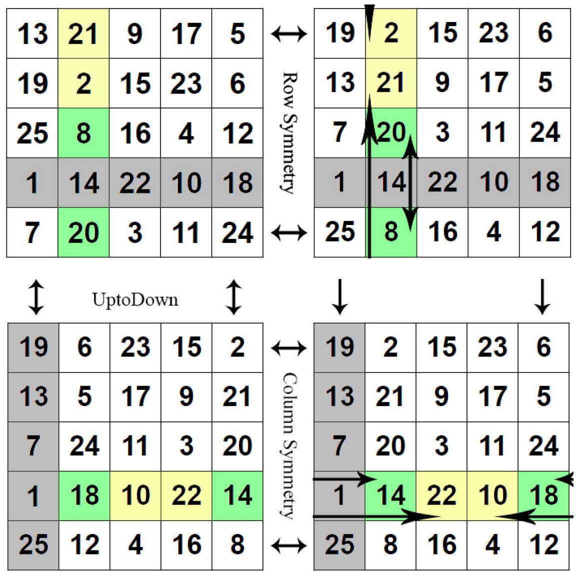
```
void RightLefttoUpDown(int** M,int i,int j){
        int ii,jj,k,ti,tj,temp;
        //Finding the fixed position
        if(i<j){
                j-=i;
                i=N-j;}
        else
                i-=j;
        //Phase 1
        for(ii=i;ii<N;ii++)
                for(jj=0;jj<ii-i;jj++){
                        temp=M[ii][jj];

M[ii][jj]=M[jj+i][ii-i];
                        M[jj+i][ii-i]=temp;
                }
        //Phase 2
        for(jj=N-i;jj<N;jj++)
                for(ii=jj-(N-i)+1;ii<N;ii++){
                        ti=jj; tj=ii;
                        for(k=0;k<i;k++){
                                ti=moveU(ti);
                                tj=moveL(tj);
                        }
                        temp=M[ii][jj];
                        M[ii][jj]=M[ti][tj];
                        M[ti][tj]=temp;
                }
}
```

- *UptoDown*

This algorithm aims to transform a pattern that has a step Up and has a certain weighting of Right steps to the pattern that has a step Down with the same weighting of Left steps and vice versa. So it will

transform RightUpUp to LeftDownDown, LeftLeftDown to RighRightUp, etc...
As done above, one can observe that those patterns are linked via a process that is broken down into two phases: Phase 1 is fixing the row containing the starting position and swapping the square by symmetry with respect to that row. Phase 2 is fixing the column containing the starting position and swapping the square by symmetry with respect to that column. And as in the RightLefttoUpDown algorithm, it is needed to link "virtual" symmetric with actual symmetric in the square.



From Right-Right-Up to Left-Down-Down and vice versa (starting position: (1,0)).

(9) UptoDown algorithm in C++:

```
void UptoDown(int** M,int i,int j){
        int jj,ii,t1,t2,temp;
        for(jj=0;jj<N;jj++){
                t1=t2=i;
                for(ii=0;ii<N/2;ii++){
                        t1=moveD(t1);
                        t2=moveU(t2);
                        temp=M[t1][jj];
                        M[t1][jj]=M[t2][jj];
                        M[t2][jj]=temp;
                }
        }
        for(ii=0;ii<N;ii++){
                t1=t2=j;
                for(jj=0;jj<N/2;jj++){
                        t1=moveL(t1);
                        t2=moveR(t2);
                        temp=M[ii][t1];
                        M[ii][t1]=M[ii][t2];
                        M[ii][t2]=temp;
                }
        }
}
```

- *UptoDown_2*

After analyzing carefully UptoDown algorithm, one can remark that it is requiring a given value to be swapped two times before it finds its final position (one swapping in the row symmetry and one swapping in the column symmetry). Both symmetries are with respect to both the row and the column to which the starting position belongs to. Therefore, we can replace both symmetries with one symmetry with respect to the starting position itself, thus achieving one swap of values before they find their final position. This algorithm will require extreme precision and agility while working with the indices as well as requiring ongoing mapping between "virtual" and actual positions.

(9') UptoDown algorithm in C++:

```
void UptoDown_2(int** M,int i,int j){
        int ii,jj,k,ti,tj,i1=i,j1=j,temp;
        int n=N/2;
        k=j-n;
        for(jj=k;jj<=j+n;jj++)

for(ii=i+(jj-k)-n+1;ii<=i+n;ii++){
                        i1=normalize(ii);
j1=normalize(jj);
                        ti=normalize(2*i-i1);
tj=normalize(2*j-j1);
                        temp=M[i1][j1];
                        M[i1][j1]=M[ti][tj];
                        M[ti][tj]=temp;
```

```
            }
        for(k=1;k<=n;k++){
            ii=normalize(i-k);
jj=normalize(j-k);
            ti=normalize(2*i-ii);
tj=normalize(2*j-jj);
            temp=M[ii][jj];
            M[ii][jj]=M[ti][tj];
            M[ti][tj]=temp;
        }
}
```

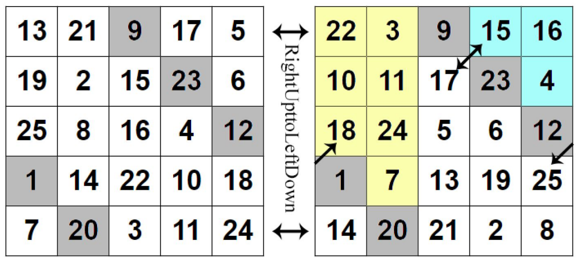
The comparison, between the complexity of this algorithm of O(n^2/2) to the one of UptoDown O(n^2), will permit to conclude that this algorithm is more efficient as it does half the swapping despite the constant check for actual positions and the complex calculations on the indices. However after real life testing, results were devastating, this algorithm turns out to be much worse than the original pattern algorithms. The answer as to why this happened takes us back to our algorithm NextRight, when it was shown that the algorithm will have much better run time efficiency then the patterns algorithm despite having the same complexity on paper. Here the same problem is faced as we are continuously reaching out to farther away elements not being accessible fast enough. This phenomenon reveals even more as to how the Cache calls elements for treatment from a given matrix stored in the RAM. A block of nearby variables is called along with the element requested and is stored in the cache. However this block is not big enough to contain the whole matrix in the cache, thus the CPU will have to wait when requesting access to another variable (as this variable is not a nearby variable of the already requested variable, therefore it is not available currently in the cache) to be called from the RAM to replace along with its block of nearby variables, the already available and accessible variables in the cache (cache memory is relatively small). This swapping of "access" to variables will keep on repeating itself, and no benefits will be taken from the availability of the block of variables in the cache, other than the requested element itself.

Given that finding, this algorithm will not be used in the final product, but it sure was beneficial in showing clearly one of the hidden criteria to an algorithm to be efficient as well as offering a new perspective on symmetry in squares and the complex play on the indices.

- *RightUptoLeftDown*

Having developed algorithms (8) and (9), looking further more into them one can merge their properties to come up with a new one. Algorithm (8) will transform a perfect magic square that was created using a pattern that weight more Right or Left to a perfect magic square having the pattern that weight more Up or Down and vice versa; and it was proven that in fact it is a symmetry with respect to the major pseudo-diagonal containing the starting position.

In the same logic, algorithm (9) will transform a perfect magic square that was created using a pattern that has a step Up and has a certain weighting of Right steps to a perfect magic square having the pattern that has a step Down with the same weighting of Left steps and vice versa. Now, if it is requested to go directly from a perfect magic square of type RightUpUp to another perfect magic square of type LeftLeftDown or vice versa, the program will need to go through two algorithms in order to deliver the request. However combining both algorithms' properties: Symmetry with respect to the major pseudo-diagonal that contains the starting position + Symmetry with respect to the starting position = Symmetry with respect to the minor pseudo-diagonal that contains the starting position.



From Right-Up-Up to Left-Left-Down and vice versa (starting position: (1,0)).

This algorithm, however, proves more challenging then the major pseudo-diagonal symmetry, as the minor pseudo-diagonals goes against how the indices are set (i from bottom to top, j from left to right). Consequently, one can observe that what characterizes each line going from top left to right bottom is in fact the sum of the indices being the same for the elements of this line. This property will be used to establish the algorithm. As in the algorithm (8), this algorithm will be split into two phases, one being a square area of the big square, and the other being the other elements that need to be swapped and mapped between “virtual” and actual elements of the square. And as a minor pseudo-diagonal is in fact the union of two lines of the type described before, we will need to start with a landmark being the topmost line part then change our landmark to fit the other line.

(10) RightUptoLeftDown algorithm in C++:

```
void RightUptoLeftDown(int** M,int i,int j){
        int ii,jj,k,ti,tj,t,temp;
        while(j>0 && i<N-1){
                i++;
                j--;
        }
        if(!j && i!=N-1)
                j=1+i;

        for(jj=j;jj<N;jj++)

for(ii=N-1;ii>(N-1)-(jj-j);ii--){
                        temp=M[ii][jj];
                        k=(ii+jj)-(N-1+j);

M[ii][jj]=M[ii-k][jj-k];
                        M[ii-k][jj-k]=temp;
                }
        t=(j+N-1)-N;
        for(jj=j-1;jj>=0;jj--)
                for(ii=j-jj;ii<N;ii++){
                        ti=ii; tj=jj;
                        k=(ii+jj)-t;
                        for(int
o=0;o<k;o++){
                                ti=moveD(ti);
                                tj=moveL(tj);
                        }
                        temp=M[ii][jj];
                        M[ii][jj]=M[ti][tj];
                        M[ti][tj]=temp;
                }
}
```

C. Some Programming Obstacles

Throughout the progress of this project, some programming obstacles have been faced:

- Lack of memory usage

After the initial tests on small squares (the dimension N ≤ 401) with static declaration of the matrix M[N][N], testing on larger dimensions was met with an obstacle. The program didn’t allow the usage of more than 2MB of memory which turned out to be the stack size of a C++ program, in other words it is the memory space that a C++ program is allowed to use statically (The stack can be furthermore expanded to be 8MB large, which was still an insufficient memory amount for this project). This obstacle was fixed with the usage of dynamic memory allocation for the Matrix (calloc). But then it was still limited to 2GB, this was caused by the Visual Studio - Visual C++ Environment compiling the executable file (.exe) as a 32bit executable by default. Modifying this setting and compiling a 64bit executable, along with dynamic allocation, there was no more limit on the memory usage of the program.

Dynamic allocation code in C++:

```
int **M;
M=(int **)calloc(N+1,sizeof(int *));
        for(int y=0;y<N;y++)
                M[y] = (int *)calloc(N,sizeof(int));
        M[N]=NULL;
```

- Low CPU usage

Running the program and opening the task manager, one can see that the program is using a maximum of 10% CPU Usage and taking a while to finish the required tasks, while there are still plenty of usage available. This problem was fixed by giving the program a higher priority class, so that the CPU allocates more run time for the program, in favor of other lower priority running programs on the computer.

Priority class code in C++:

```
#include <windows.h>
DWORD dwError;
if(!SetPriorityClass(GetCurrentProcess(),HIGH_PRIORITY_CLASS)){
                dwError=GetLastError();
                cout<<"Failed to get priority \n"<<dwError<<endl;
        }
```

IV. Results

All test cases have been conducted on a HP Envy 17” with a Core i7 CPU having the following specifications:

Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz

Base speed:	*2.40 GHz*
Sockets:	*1*
Cores:	*2*
Logical processors:	*4*
Virtualisation:	*Disabled*
Hyper-V support:	*Yes*
L1 cache:	*128 KB*
L2 cache:	*512 KB*
L3 cache:	*4.0 MB*

Note that the UptoDown algorithm could be further more optimized by splitting it in two phases just like the other algorithms. In fact, it has no reason to check for out of range indices in each iteration step. Indeed, it will be kept like so, to show even more clearly how reaching out to farther away elements of a matrix (pattern algorithm, ex: RightUpUp) is even worse than continuous calls to other functions and tests on the values.

- *Test 1*

This test case aims to show the performance difference between the different space filling pattern algorithms: Right-Up-Up, Right-Right-Up, Left-Down-Down and Left-Left-Down.

- *Test 2*

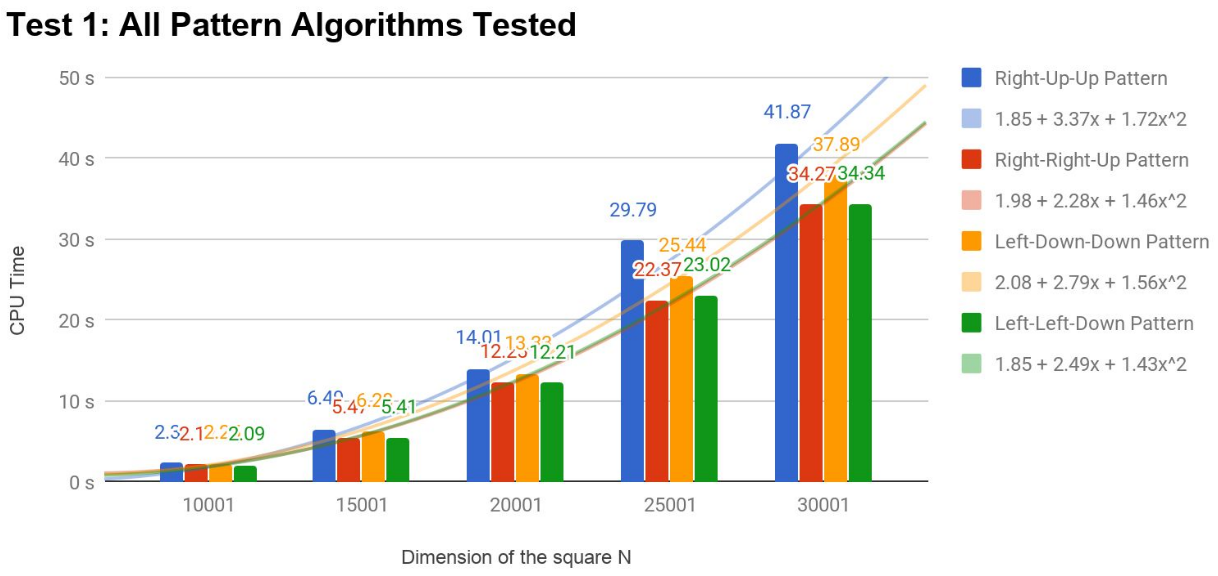
This test case aims to show the performance difference between the space filling pattern algorithms (ex: Right-Up-Up) and the NextRight and NextUp algorithms.

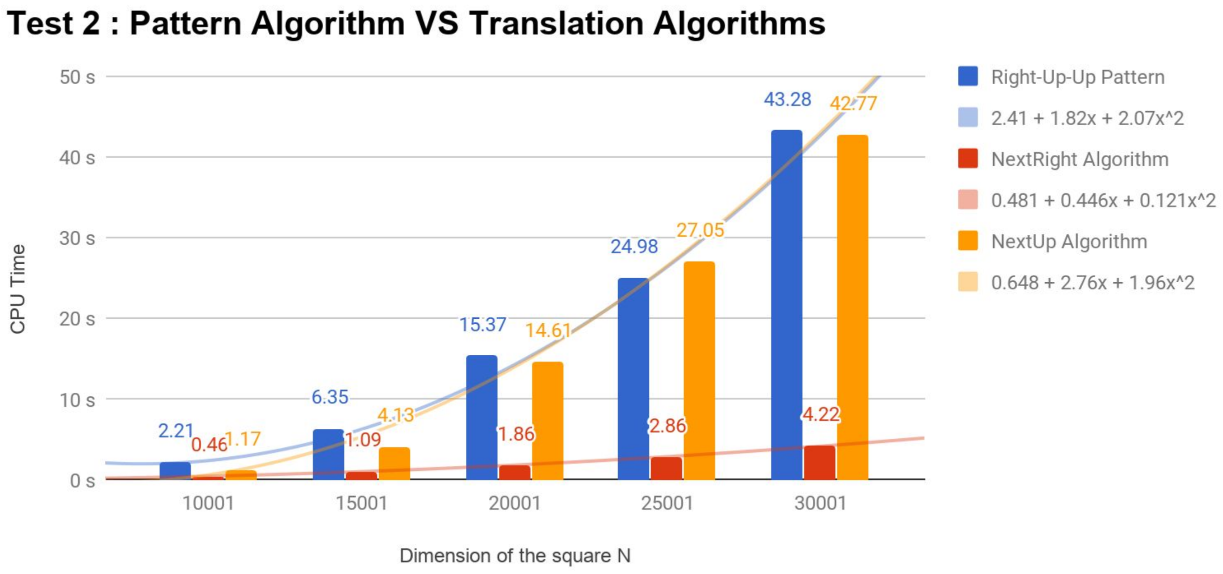
- *Test 3*

This test case aims to show the performance difference between the space filling pattern algorithms (ex: Right-Up-Up) and developed mapping algorithms.

- *Test 4*

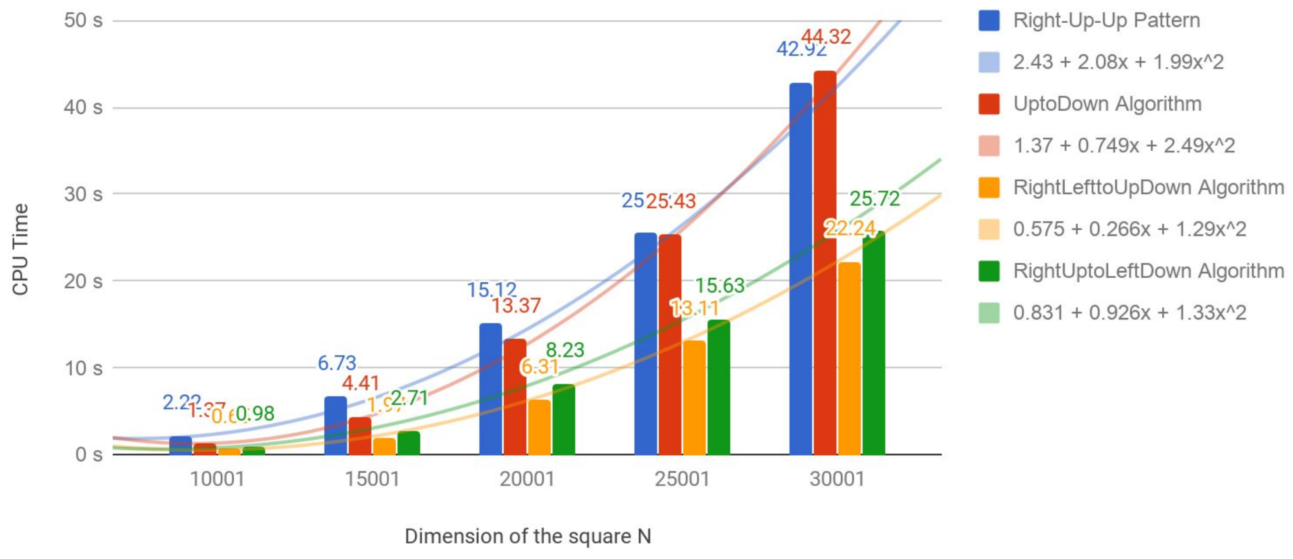
This test case aims to show the performance difference between different algorithms combinations. to produce all $4*N^2$ combinations of a given dimension N.



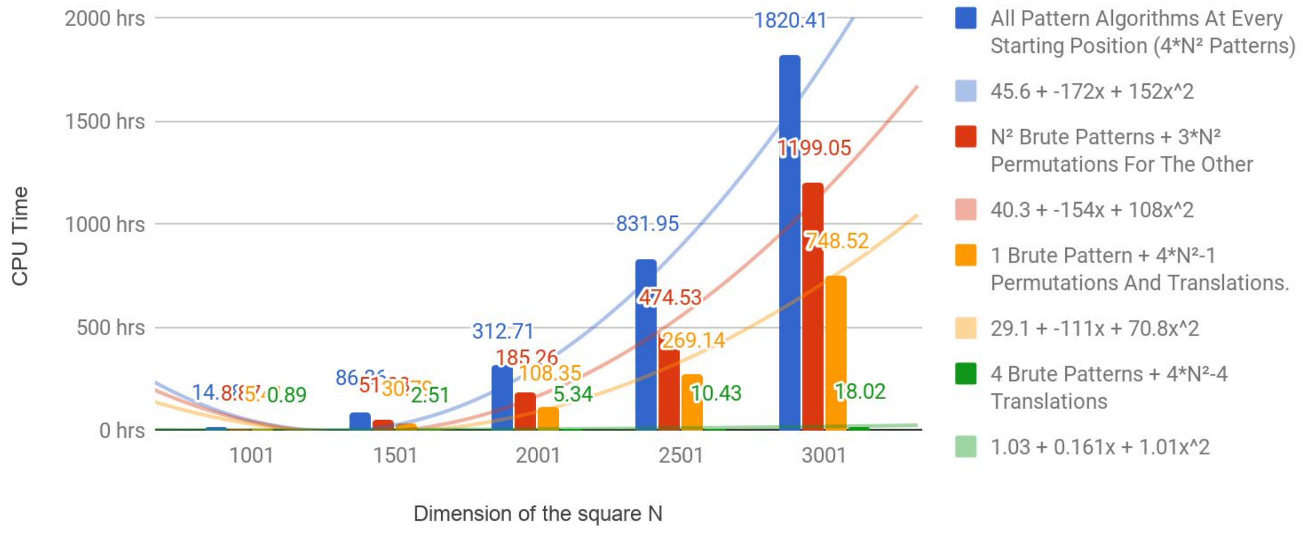


In the test case 1, we can observe that the pattern algorithms moving two steps either Up or Down, have a lower performance compared to the pattern algorithms moving only one step either Up or Down. This goes to confirm the previous conclusion that the access to the next column is faster then the access to the next row. Furthermore the pattern algorithms with a step Up perform better than the pattern algorithms with a step Down, which means that the access to the previous row is faster then the access to the next row. In the second test case, We see the damage of the latter discussed access time difference, as the NextUp algorithm slowly loses its efficiency with higher dimensions, to reach a point where it is no longer beneficial in terms of replacing the original pattern algorithms. However the NextRight algorithm keeps its good efficiency even at higher dimensions.

Test 3: Pattern Algorithms VS Mapping Algorithms



Test 4: Generation of all 4*N² Combinations of Perfect Magic Squares for a given Dimension



In the test case 3, the results confirm our thoughts at the beginning of this section. We can clearly see the inconvenience of constantly calling functions, running test on values and returning values. However the inconvenience of reaching out further away elements is worse or the same after reaching higher dimensions. In the final test case, we clearly see the benefits and the results of the optimization. First running the four pattern algorithms at each starting position, then running a pattern algorithm once at every position and mapping to the others, after that we only called a pattern algorithm once and generated all the other combinations via mapping and translation; finally we ran the four pattern algorithms, each once, and translated to find the other combinations.

V. Conclusion

In conclusion, this work has helped in developing a sense of analytical thinking, offering new perspectives on squares and patterns properties, as well as enhancing algorithmic skills and agility. Throughout this work many performance criteria have been taken into consideration to optimize the various algorithms, and most notably the time to access a given variable from a square; which helped understand how the cache and the RAM communicate and transfer data in between them and how the square is stored. More generally learning when and why accessing some variables is faster then accessing others. Indeed the results section confirm all analysis done in this work as the progression of the CPU Time is polynomial of degree 2, and not linear, caused by the "access" factor.

After going through all the algorithms and the tests results, we can give some system recommendations as to what algorithms to use depending on your machine hardware specifications, and your needs. Having a lot of memory, createAll algorithm is for sure the best choice. Having more CPU power and you'd be better using the other mapping and translation algorithms. Working with small dimensions all algorithms perform well. However when the dimensions get higher, it is recommended to avoid using NextUp.

Finally, an aperture of this work, is to study even dimensions magic squares and their specific algorithms and properties. Another more interesting topic, would be the study of a parallel algorithmic analysis that uses HPCN (High Performance Computing Network) framework with the implementation under MPI-PVM (Message Passing Interface - Parallel Virtual Machine) which will improve the performance of such work and will allow the computation of bigger dimensions squares, as well as significantly improving the CPU Time.

Personally, pursuing my graduate studies next year at the ULFS-II i would be much interested in the continuation of this work, as described earlier, as i chose this subject with continuation objectives in mind and i see much potential benefits from such work that can be further and further explored.

VI. References

[WM001] Wolfram-Mathworld "Magic square". http://mathworld.wolfram.com/MagicSquare.html

[WK001] Wikipedia "Lo Shu Square". https://en.wikipedia.org/wiki/Lo_Shu_Square

[WK002] Wikipedia "Magic square" 3.3 Some famous magic squares - Albrecht Dürer's magic square. https://en.wikipedia.org/wiki/Magic_square

[AR001] Weibin Z, Yu Hui D, Kai-Tai F "Image encryption by using magic squares" 2016. https://ieeexplore.ieee.org/document/7852813/

[AR002] Parth P "Image Encryption Algorithm Using Ancient Magic Squares" 2011. https://www.researchgate.net/publication/281001870_Image_Encryption_Algorithm_Using_Ancient_Magic_Squares

[AR003] Peyman F, Ramin J "An Introduction to Magic Squares and Their Physical Applications" 2015. https://www.researchgate.net/publication/297731505_An_Introduction_to_Magic_Squares_and_Their_Physical_Applications