Construction of random magic square

Introduction

In this paper I will show you some methods for building random generated magic square with a computer program.

From an historical view, the program was born for generating magic square of consecutive primes numbers ("*Prime Magic Square*"). However, you can give whatever sequences of input values you want and the program will calculate the possible magic constant and build the relative square if possible (so far it was used even with *Smith* numbers, but you can use even for generation normal magic square).
Else the program can generate square even for big orders (so far it was coded for order up to 63, but it needs only to code the relative big order to use) and in very short time.

From a statistic analysis, the program is able to give result if this condition is reached:

Let:

- A1, A2, … An*n the input sequences of values for a square of order N
- MAGIC = magic constant of square with A1..An*n values
- Cn*n,n = number of combination of N elements taken from N*N set of numbers
- COUNT=number of possible sequences of N numbers from N*N set of numbers that have a sum of MAGIC

If

- COUNT > 1% Cn*n,n

Then the program can give a square with a very high probability. More big is COUNT related with Cn*n,n and more easily a square can be built.

So, for a order 6, we have:
Cn*n,n=1947792
That means that COUNT must be something like 19000 for having a good probability to get a square

Method of construction

The program actually had 4 kind of different methods for magic square generation, and each one has some variants to use.

Here I show a table with the available methods and then I will describe each method in details (please, note the some methods can no give result depending by the order and input sequence, but at least one of them is able to build a magic square):

Method	Description	Version
A1	N rows ‖ N columns ‖ 2 diagonals ‖ N rows	V1
A2		V2
A3		V3
B1	2 rows ‖ 2 diagonals ‖ N rows ‖ N columns	V1
B2		V2
B3		V3

C1	N rows || N diagonals || 1 diagonal	V1
C2		V2
C3		V3
D1	N rows || N diagonals || 1 diagonal || 1 diagonal	V1
D2		V2
D3		V3
D4		V4

At the beginning of program, not depending from the method to apply , it follows those steps:

- Reads the sequences of N*N elements. N is unknown and will be determine during this step.
- Calculates the magic constant of this square by summing all the N*N elements and dividing the result for N. If the result has no rest, then this number is a valid one (there can be another condition for prime numbers, like that magic constant must with the same parity of order for being accepted, but this rules was relaxed for allowing to have a free input sequences).
- Random puts all the N*N elements into the square

- Now, the given selected method will be apply to the square.

Method Ax (x=1, 2, 3)

This method can be describe as follow:

1. make all rows magic
2. rotate the square
3. make all rows magic (that was columns in point 1)
4. make diagonals magic using two magic rows
5. remake all rows magic without changing diagonals state
6. rotate the square
7. remake all rows (columns in point 5) magic without changing diagonals state

This is done using version *x* in some points

Method Bx (x=1, 2, 3)

This method can be describe as follow:

1. make 2 rows magic
2. make diagonals magic using two magic rows
3. remake all rows magic without changing diagonals state
4. rotate the square
5. remake all rows (columns in point 3) magic without changing diagonals state

This is done using version *x* in some points

Method Cx (x=1, 2, 3)

This method can be describe as follow:

1. make all rows magic
2. rotate the square
3. make all rows magic (that was columns in point 1)
4. make one diagonals magic using permutation (and pray for the other)

This is done using version *x* in some points

Method Dx (x=1, 2, 3, 4)

This method can be describe as follow:

1. make all rows magic
2. rotate the square
3. make all rows magic (that was columns in point 1)
4. make diagonals magic using two algorithms
5. remake all rows magic without changing diagonals state
6. rotate the square
7. remake all rows (columns in point 5) magic without change diagonals state

This is done using version *x* in some points

Detail description

At this point I will explain how each of the above points are implemented into the program.

The first routine we analyze is a simple one, but one of the most used. It is called *swapping1* into the source.

The porpoise of this routine is very simple: given two rows into the square, make the first row magic by swapping some elements from the second row at the same column index position. Essentially, it swap for all the combination elements from the two rows, until the first become magic. If the row not becomes magic, then this routine fails.

Here an example in a 5x5 square with some Ax and Bx numbers:

A1	A2	A3	A4	A5
B1	B2	B3	B4	B5

(A1+A2+A3+A4+A5) != MAGIC AND (B1+B2+B3+B4+B5) != MAGIC

A1	B2	A3	B4	B5
B1	A2	B3	A4	A5

(A1+B2+A3+B4+B5) = MAGIC

A similar routine is called *swapping2* into the source and make the same thing of the previous routine with one exception: no elements that are into the diagonal position are swapped (red element into the example square).

A1	A2	A3	A4	A5
B1	B2	B3	B4	B5

(A1+A2+A3+A4+A5) != MAGIC AND (B1+B2+B3+B4+B5) != MAGIC

A1	B2	B3	A4	B5
B1	A2	A3	B4	A5

(A1+B2+B3+A4+B5) = MAGIC

The fact that there is this two kind of different routines will be evidence now that we analyze some important routines:
intellSwap1, *intellSwap2, intellSwap3*, *intellSwap4*

The scope of those routines is simple: make all rows of the square to the magic constant value.

The common routine is this:

- Scan all the rows from up to down
  - Scan all the rows under the selected one
    - Make the first selected row magic by *swapping1* or *swapping2* with the second selected row

As you can see, all rows are made magic just with swapping with elements of the other rows at the same column position, that means that in all those swaps, the magic value of sum of column are not changed.

After a calling to this routine all that we have is all rows made to magic state and columns state not changed, while the state of the diagonals depends by the swapping routine used:

- *intellSwap1* use *swapping1* so diagonal state are changed after his execution
- *intellSwap2* use *swapping2* so diagonal state are not changed after his execution

One point to see is that when it is make the N-1 row magic using the N row, what we get is that even N becomes to a magic state (at least in 99,9% of cases as with Smith numbers it seems to fails in some cases).

Those routines, used into all the *Ax*, *Bx*, *Cx* and *Dx* methods, works well, but when the order is increasing and so there is a remarkable difference from two numbers, it can be less efficient when it manages the last rows of the square.

A solution I see when coding them is to choose a swapping combination only if the sign of difference to the magic constant is different from the one in the first row respect the second one.

This assumption let be the state of one row near to the magic state after a swap, otherwise the difference will increase and make hard to find a solution for the other remaining rows.

Here and example

A1	A2	A3	A4	A5
B1	B2	B3	B4	B5
C1	C3	C3	C4	C5

(A1+A2+A3+A4+A5) > MAGIC AND (B1+B2+B3+B4+B5) > MAGIC AND
(C1+C2+C3+C4+C5) < MAGIC

So, this swap is not done, as second row will have a much high difference with MAGIC:

A1	B2	B3	A4	B5
B1	A2	A3	B4	A5

(A1+B2+B3+A4+B5) = MAGIC AND (B1+A2+A3+B4+A5) >> MAGIC

Instead this swap is performed:

C1	A2	A3	C4	A5
A1	C2	C3	A4	C5

(C1+A2+A3+C4+A5) = MAGIC AND (A1+C2+C3+A4+C5) more near to MAGIC

So, the other two routines operate as follow

- *intellSwap3* use *swapping1*so diagonal state are changed after his execution, but swaps only for reducing the difference from the magic value in other rows
- *intellSwap4* use *swapping2*so diagonal state are not changed after his execution, but swaps only for reducing the difference from the magic value in other rows

At this point it is evident that if a square is rotated and then one of the above routines is used for making rows to magic state, it is the same of having make the columns magic into the original square, as no elements in columns are changes when operating to the magic state of rows.

Now, we are near to understand how methods *Ax* and *Bx* work, but before we have to describe another important routine: *diagonal1*.

diagonal1 is a routine that makes the two diagonal to magic state in very simple manner. Historically, *Method Ax* e *Bx* was born after *Method Cx*, where the problem of making diagonals to magic was very difficult, but the simple way it operates into the *Method Ax* and *Bx* will for sure

make you smile, as the global result is perform with less problems that in *Method Cx* and with good result! I will return to this point soon after the description of diagonal process.

If you look to the simple description of *Method Ax*, *Cx*, and *Dx*, all that you see is the common structure that is:

- Make all rows magic
- Make all columns magic
- Make the two diagonals magic.

Even if trying to make diagonals magic having all the square to magic state seems simple, it is the most difficult task, has you cannot make too many swap without breaking rows and columns magic state.

The simple solution used by *diagonal1* was this: make diagonals to magic state breaking rows and columns magic state as you need!!

In fact, it operates as this:

- For even orders:
  - Copy the first row to the diagonal \ and the last row to diagonal /
  - So, this works only good if first row and last row are magic.
  - If columns were magic, this don't change the magic state, while if rows were magic, they will become not magic.
- For odd orders:
  - Copy all elements of row in half position to the diagonal \ and all the elements of column in half position to diagonal /
  - This changes the magic state of rows and columns

So, essentially, one row and one column are placed into the diagonals for making them magic.

Example for Even:

A1	A2	A3	A4	A5	A6
B1	B2	B3	B4	B5	B6

A1					B6
	A2			B4	
		A3	B4		
		B3	A4		
	B2			A5	
B1					A6

Example for odd:

		B1		
		B2		
A1	A2	AB	A4	A5
		B3		
		B4		

A1				B4
	A2		B3	
		AB		
	B2		A4	
B1				A5

Now we have all elements for looking the detail implementation of *Method Ax*

Remember so the first simple definition:

1. make all rows magic
2. rotate the square
3. make all rows magic (that was columns in point 1)
4. make diagonals magic using two magic rows
5. remake all rows magic without change diagonals state
6. rotate the square
7. remake all rows (columns in point 5) magic without changing diagonals state

Method A1:

1. *intellSwap1*
2. rotate square
3. *intellSwap1*
4. *diagonal1*
5. *intellSwap2*
6. (only for odd order) rotate the square
7. (only for odd order) *intellSwap2*

Method A2:

1. *intellSwap3*
2. rotate square
3. *intellSwap3*
4. *diagonal1*
5. *intellSwap4*
6. (only for odd order) rotate the square
7. (only for odd order) *intellSwap4*

Method A3:

1. *intellSwap3 intellSwap1*
2. rotate square
3. *intellSwap3 intellSwap1*
4. *diagonal1*
5. *intellSwap4 intellSwap2*
6. (only for odd order) rotate the square
7. (only for odd order) *intellSwap4 intellSwap2*

Looking at the details it is evident that *Method A3* is better that *Method A2* that is better that *Method A1*, but *Method A1* should be fast in some conditions. *Method A3* is like *Method A2*, but if it fails in making some rows/columns magic, it try with the swap of *Method A1*.

Now you can see how simple is this method of solution: it takes two row/column for making the diagonal magic, braking all the rows and columns magic state, but after, it rebuilds the magic state off all rows and columns without changing the magic state of diagonal. When I code this, I cannon expected to have results, but instead it works perfectly.
I so try to make *Method Ax* another step versus simplicity by creating *Method Bx*, that we will see now. It is important to look that even if *Method Bx* seems to not give speed result when the other methods are very fast, it gives a very good rate with big orders, when the other methods are slow (this is quite surprising!).

Here what was our simple definition of *Method Bx*

1. make 2 rows magic
2. make diagonals magic using two magic rows
3. remake all rows magic without change diagonals state
4. rotate the square
5. remake all rows (columns in point 3) magic without changing diagonals state

The idea now is simple: first make the two diagonals to the magic state (and this is done by making two rows magic and copy them to the diagonals) and then make all rows and column to the magic state without changing the diagonal state.
The fact that we make the point 3 and 5 without changing the diagonal state is what make this algorithm potentially less fast than the others, as while not moving elements in diagonal reduce the space of possible moving. However with order increasing maybe the space of moving is big and so this program, that has less operation of the others, can do his best.

The only point to see about description of *Method B1*, *B2* and *B3* is that they are identical to *Method A1*, *A2* and *A3*, but with the difference in the first steps that they starts to make diagonals magic by coping by two rows (that are first and last row for even order and row at half position and column at half position for odd order).

Now we look at the first type of program I make for generating the prime magic square: *Method Cx*.

If we look at initial simple definition, we have:

1. make all rows magic
2. rotate the square
3. make all rows magic (that was columns in point 1)
4. make one diagonals magic using permutation (and pray for the other)

As you can see this is the logical process for making a magic square: make all rows to magic state, then make all column to magic state and finally make the two diagonals to magic state without braking the magic state of rows and columns.

This is instead the detail definition:

Method C1:
1. *intellSwap1*
2. rotate square
3. *intellSwap1*
4. *diagonal2*

Method C2:
1. *intellSwap3*

2. rotate square
3. *intellSwap3*
4. *diagonal2*

Method C3:

1. *intellSwap3 intellSwap1*
2. rotate square
3. *intellSwap3 intellSwap1*
4. *diagonal2*

Now all should be clear for point 1 to 3, as identical to the other in *Method A1*, *A2* and *A3*.

The difference is done only by *diagonal2* routine that now we will describe.

The idea behind *diagonal2* is simple:

- Permute the position of the rows (note that this not change the state of columns magic state) for making diagonal \ to magic state until even diagonal / comes to magic state.

As all permutation are calculated for this step, this means that with high order this routine can take a long time to run (even if it is not necessary it takes all permutation if the space of possible combinations is big, as in initial statistic analysis).

For preventing that too many times is given to a magic square that has not found a solution in short time, Method C3 has an option that stop diagonal permutation if it has made some passed max number of steps. With this option a square can be found in less time even for some high order (it uses *diagonal3* instead of *diagonal2*, and *diagonal3* has a time out function).

Here a simple example of how diagonal process works (in red the choose numbers that make the magic constant after the rows rearrangement):

2	13	8	11
16	3	10	5
7	12	1	14
9	6	15	4

Now swaps row 1 in row 2 and row 2 to row 3 for having the right diagonal \ state

7	12	1	14
2	13	8	11
16	3	10	5
9	6	15	4

In this case even diagonal / becomes magic.

Now we have the latest method of solution to analyze that it is just a step over the *Method Cx*, and a little below *Method Ax*.

1. make all rows magic
2. rotate the square
3. make all rows magic (that was columns in point 1)
4. make diagonals magic using two algorithms
5. remake all rows magic without changing diagonals state
6. rotate the square

7. remake all rows (columns in point 5) magic without change diagonals state

The difference here is that when making diagonals to magic state, two algorithms are used, one that not changes rows/columns magic state and one that changes it, so after that it is necessary to remake rows and columns magic again (however due to changes made in diagonals, columns magic state are not changed and so steps 6 and 7 are not to be performed).

Here a detail description about the methods:

Method D1:
1. *intellSwap1*
2. rotate square
3. *intellSwap1*
4. *diagonal4*
5. *intellSwap2*

Method D2:
1. *intellSwap3*
2. rotate square
3. *intellSwap3*
4. *diagonal4*
5. *intellSwap4*

Method D3:
1. *intellSwap3 intellSwap1*
2. rotate square
3. *intellSwap3 intellSwap1*
4. *diagonal4*
5. *intellSwap4 intellSwap2*

Method D4:
1. *intellSwap3 intellSwap1*
   - rotate square
   - *intellSwap3 intellSwap1*
   - rotate square
   - *intellSwap3 intellSwap1*
2. rotate square
3. *intellSwap3 intellSwap1*
4. *diagonal4*
5. *intellSwap4 intellSwap2*

The only remarkable difference here is to shows that *Method D4* is like *D3*, but with the difference that if we are not able to make all rows to magic state, we try to make columns to magic state and then retry with rows again.

Finally we have to speck about *diagonal4* works: it makes the diagonal \ to magic state without changing rows and columns magic state (by swapping rows positions), then makes the other diagonal / to magic state changing rows magic state.

The first step is achieved by rearrange rows like we see for *diagonal2* and the routine stop as soon as a permutation that make diagonal / to magic state is fount.

The second step is achieved by founding some swaps of cells (in rows) that not touch element into diagonal \ , but make diagonal / to magic state (essentially we brake rows magic state for make diagonal \ to magic state)

Final consideration

The source code of those programs is released as GPL here and are C++ programs:
http://digilander.libero.it/ice00/magic/general/download.html

They have a common part and custom header for managing each orders: you can easy code the order you need by modify a given header.