Sliding Pieces moves generation without rotated BitBoard.

By Felice Pollano

www.felicepollano.com

February, 18 2007

This article describe the way I used in writing a trivial but funny chess engine in C#. In the traditional approach, for any rank, file and diagonal in the chess board, we consider an occupancy value that is actually the integer representing all the pieces of any color laying on a certain rank or file or diagonal. So we can create four arrays in memory (ranks, files, two diagonal), indexed by both the occupancy number and the square number pointing to a bitmask of legal moves and defenses (discriminating a real move from a defense is done on move generation) . Due to the fact that computing rank occupation is only a matter of shifting, but files and diagonal is more complex, programmer uses some sort of rotated bitboard, to make all computation the same case of rank. This involves keeping aligned in memory (at least) four different bitboard, rotated at 0,90,45,-45. Furthermore computing the occupation is still not immediate, even if very simple.

64 bit integers as occupancy values.

Sliding part can move both horizontal and vertically, and 45-145 degree in diagonal. So we consider each cell in the bitboard (64) , and for each cell we consider four arrays:

- Diag45: for each square on the board, point to an integer having '1' in each square lying on the diagonal the cell we consider, except the square itself.
- Diag135: the same as above, but for the opposite diagonal.
- File: The same as above for files
- Rank: The same as above for ranks.

Let's have some example:

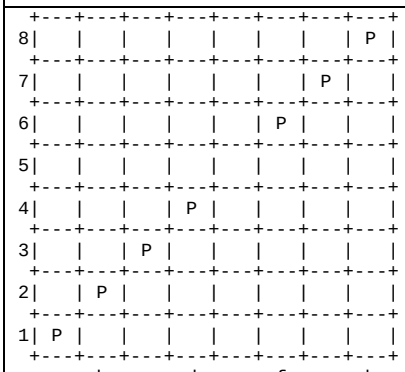
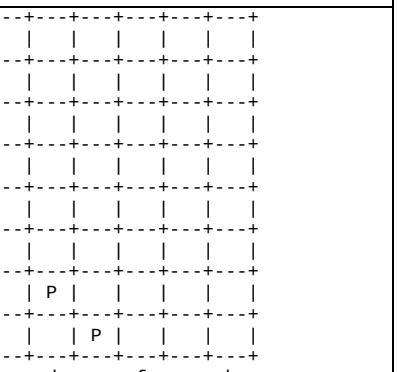
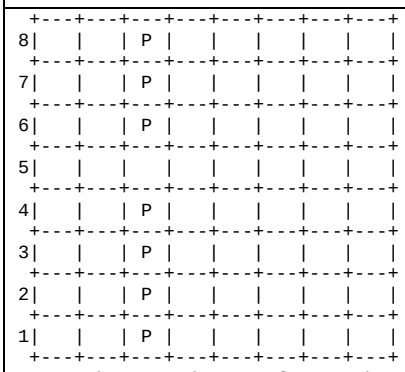
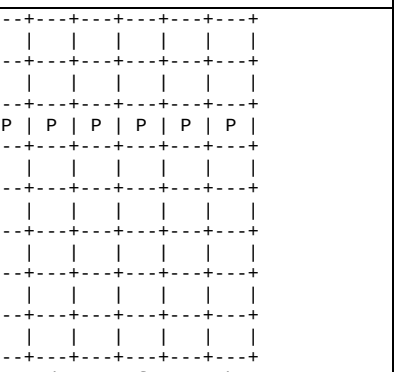
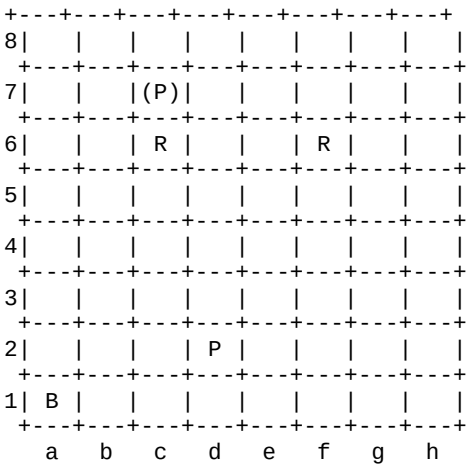
Diag45 for e5	**Diag135 for c3**
	
Diag45[28] = **0x0102040800204080**	Diag135[42]= **0x1008000201000000**
File for c5	**Rank for b6**
	
File[26] = **0x0404040400040404**	Rank[17] = **0x0000000000fd0000**

Table 1

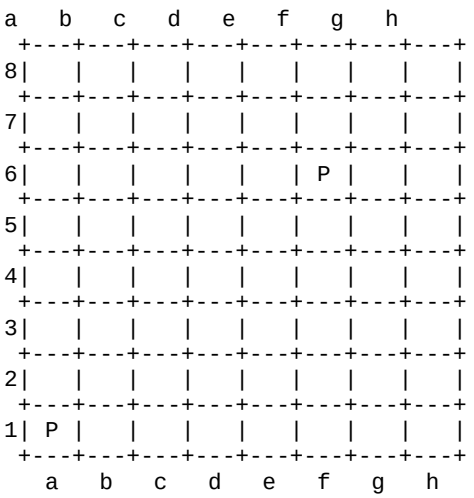
In the table above we have the four array taken from sample cells. For each square in the “live” board in the game, **we can compute an occupancy degree**, represented by an integer 64, for file, ranks and the two diagonal. This is done **by simple bitwise AND the “All pieces” bitboard whit the value found in the respective table**.

Let’s have an example. Consider an example board partially filled with pieces like the following one:



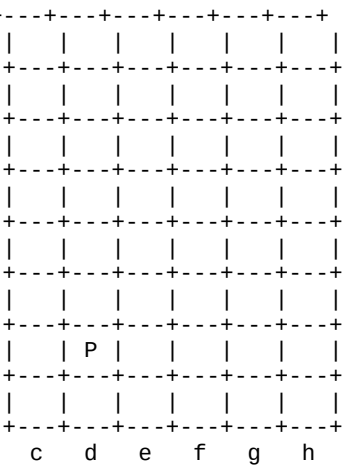
Of course the board above does not represent a real game situation, but is just for clarify. We now compute and show the result of the occupancy for the cell. Please refer to Table1 for the array references.

e5 for the 45 degree diagonal. This is done by taking the bitwise and of the Diag45[28] and the bitmask of all pieces in the board we consider. We obtain:



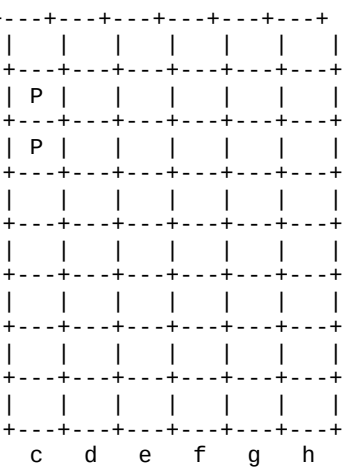
and the occupation value is: **0x0100000000200000**

c3 for the 135 degree diagonal. This is done by taking the bitwise and of the Diag135[42] and the bitmask of all pieces in the board we consider. We obtain:



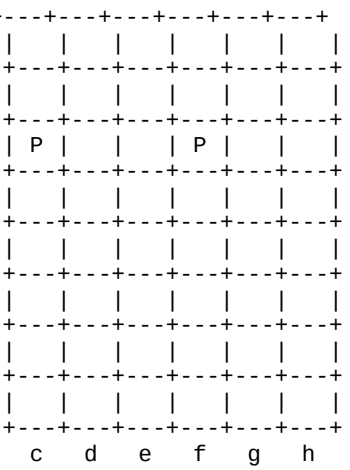
and the occupation value is: **0x0008000000000000**

c5 for the file. This is done by taking the bitwise and of the File[26] and the bitmask of all pieces in the board we consider. We obtain:



and the occupation value is: **0x0000000000040400**

b6 for the rank. This is done by taking the bitwise and of the Rank[17] and the bitmask of all pieces in the board we consider. We obtain:



and the occupation value is: **0x0000000000240000**

As a result, we have stored four pre calculated arrays each one of 64 elements. With a simple bitwise and we can calculate at any time a 64 bit integer representing the file, rank or diagonal occupation for a piece laying in a certain square by just using a fast bitwise and operation.

Using dictionaries to store pre calculated moves

Of course we cannot use the raw occupancy value as an index for an array containing all the possible moves of a piece. This just because we cannot have an array with an upper bound of MaxInt64. On the other hand, for a certain square on the board, we can have only some occupation values. So we can imagine to use a dictionary to map this finite set of values to the corresponding bitmask of available moves. Please refer to Table1, diagram **Diag135 for c3.** We have sixteen (and no more) possible occupation configuration, we show these in the table below:

Table of possible occupation value in 135 degree diagonal for square c3

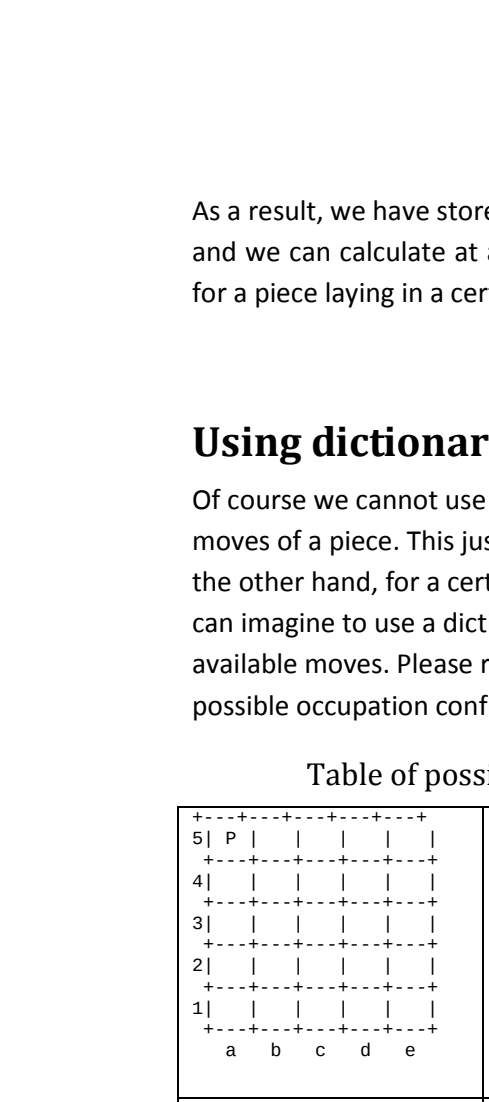
			

Table 2

So, for cell c3, we have the 16 occupation configuration available as shown above. We can represent each of these configuration, as always in bitboard strategy, by 64 bit integers. We obtain these values:

```
0x 1000000
0x 200000000
0x 201000000
0x 8000000000000
0x 8000001000000
0x 8000200000000
0x 8000201000000
0x 10000000000000000
0x 10000000001000000
0x 10000000200000000
0x 10000000201000000
0x 10080000000000000
0x 10080000001000000
0x 10080000200000000
0x 10080000201000000
0
(Table 3)
```

So we just have to calculate the available moves for each of these occupation configuration, and store in a dictionary for further retrieve. We have to consider the fact that available moves must contains the ones who interfere with the first obstacle in any direction. This just because the occupancy does not take in account the piece color. The programmer will bitwise 'and' the results bitmask of moves with the all pieces of the opponent color or empty squares to obtain realizable moves, and with its own color to check how part is defending. We will show better this later.

Storing pre calculated dictionaries

We have four different moving strategy: rank, file, 45 degree diagonal and 135 degree diagonal. We have as well 64 squares on the board. So we need four array with 64 entry each one. Each entry in the array will point to a dictionary, indexed by the occupancy number and exposing the available moves bitmask. We can store these data in a file, but in the engine I wrote I prefer to store the dictionaries as static data. I used the CodeDom feature available in the .NET framework to create an automatic tool that wrote for me the code (basically a lot of number). The following examples are extract of these generation routines that actually are implemented as unit test in NUnit.

Just to clarify the structure of the array of dictionary, I report below the declaration in C#:

```
private static System.Collections.Generic.Dictionary<ulong, ulong>[] moveHashTable;
```

the declaration shown apply for all the four case discussed (rank, files, 45 degree diagonal and 135 degree diagonal). Always as an example, some of the code used to fill the array:

```
moveHashTable = new System.Collections.Generic.Dictionary<ulong, ulong>[64];
                moveHashTable[0] = new System.Collections.Generic.Dictionary<ulong, ulong>();
                moveHashTable[0][0ul] = 9241421688590303744ul;
                moveHashTable[0][512ul] = 512ul;
                moveHashTable[0][262144ul] = 262656ul;
                moveHashTable[0][262656ul] = 512ul;
                moveHashTable[0][134217728ul] = 134480384ul;
                moveHashTable[0][134218240ul] = 512ul;
                moveHashTable[0][134479872ul] = 262656ul;
                moveHashTable[0][134480384ul] = 512ul;
                moveHashTable[0][68719476736ul] = 68853957120ul;
```

…..

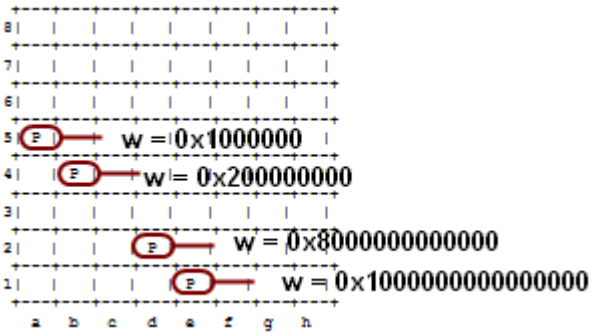
……

…..

It make no sense to report here the complete code, just because is massive and a completely repetitive. We better show some code snapshots from the unit test I used to generate these values.

Calculating the possible occupation value for a square

The idea is to find the weight of each of the bit contained in the Diag45-Diag135-Rank and File array for each cell. Then we combine these weight using a counter from 0 to the max integer that we can be represented by a number having as many digit as the number of 'ones' bit contained in the occupation value. For instance, with reference to table1, in the Diag135 for c3 we have 4 bit to take in account. We first compute the weight of these bit:



The weight are straight forward to obtains. Just shift a bit from LSB to MSB and we achieve the results. In the board representation used LSB is the square a8 and MSB the square h1.

In this situation the number of 'ones' bit is 4. The biggest number we can write using n bit is 2 raised to n:

```
combinator = Math.Pow(2, numer_of_ones);
```

In the example, we have 2^4=16, so 16 is the number we use for combinations. So we can compute all the possible occupation values by combining the weight as in the table below:

combinator	W1	W2	W3	W4	occupation
0	0	0	0	0	0
1	1	0	0	0	0x 1000000
2	0	1	0	0	0x 200000000
3	1	1	0	0	0x 201000000
4	0	0	1	0	0x 8000000000000
5	1	0	1	0	0x 8000001000000
6	0	1	1	0	0x 8000200000000
7	1	1	1	0	0x 8000201000000
…	…	…	…	…	
…	…	…	…	…	
15	1	1	1	1	0x 10080002010000000

We basically uses the 'ones' status of an incremental variable from 0 to combinator, and for each value we compute the weight by adding the weight if the respective bit in the incrementing variable is set. As we can see the results map exactly on what we show in the table 3 before.

Below the c# snap of code for generating the occupation for the 135 degree diagonal on the square 'square':

```
private ulong[] Diag135Possible(int square)
 {

    List<ulong> results = new List<ulong>();
    List<ulong> w = new List<ulong>();
    ulong diagOcc = BitBoard.Diag135(square);
    ulong p = 1UL;
    for (int q = 0; q < 64; ++q, p <<= 1)
    {
        if ((p & diagOcc) != 0)
            w.Add(1UL << q);
    }
    int combinator = (int)Math.Pow(2, w.Count);


    for (int j = 0; j < combinator; ++j)
    {
        ulong res = 0;
        if (0 != (j & 1))
            res += w[0];
        if (0 != (j & 2))
            res += w[1];
        if (0 != (j & 4))
            res += w[2];
        if (0 != (j & 8))
            res += w[3];
        if (0 != (j & 16))
```

```
                res += w[4];
            if (0 != (j & 32))
                res += w[5];
            if (0 != (j & 64))
                res += w[6];
            if (0 != (j & 128))
                res += w[7];
            results.Add(res);

        }
        return results.ToArray();
    }
```

The BitBoard.Diag135 simply accede the array we referenced in Table1 before. The functions returns an unsigned long array of occupancy value.

Calculating the possible moves from a square considering occupation

Calculating the movement is basically trivial, just slide one bit until it remains on the rank – file or diagonal we choose, in both directions, and bitwise or on the results variable. The sliding has to stop **after** we touch a piece. This just because 'touching' another piece could be a capture, if the piece is owned by the opponent, or a defense if the piece is the same color of the moving one. We don't have to care to much at the efficiency in moving computation, because this is done in pre calculation time. As a sample below I report the moving calculation for the 45 degree diagonal.

```
    private ulong GetMoveByOccupation45(int sq, ulong occ)
    {
        ulong res = 0;
        ulong square = 1UL << sq;

        ulong t;
        for (int i = 1; i < 8; ++i)
        {
            t = square >> (7*i);
            if (0 == (t & BitBoard.Diag45(sq)))
                break;
            res |= t;
            if ((res & occ) != 0)
                break;
        }
        t = square;
        for (int i = 1; i < 8; ++i)
        {
            t = square << (7*i);
            if (0 == (t & BitBoard.Diag45(sq)))
                break;
            res |= t;
            if ((res & occ) != 0)
                break;
        }

        return res;
    }
```

The function returns a bitmask in which every bit set to one represents a valid location for a piece laying on the square 'sq'.

Using the dictionary for obtaining the part movements

At that point we have four array indexed by the square (0-64) each cell pointing to a dictionary indexed by the occupancy value of the rank-file-diagonal.

Let's call these arrays

Diag135Moves

Diag45Moves

FileMoves

RankMoves

We look at the square the part we have to move is, and we found the board occupancy depending on how the part can move. So we consider rank and files for rooks, the two diagonal for bishops, and the whole for the queen. Below some example code:

```
public ulong GetRookMovesAndDefense(int sq)
{
    ulong r1 = rankmoves[sq][OccupancyRank(sq)];
    ulong r2 = filemovesm[sq][OccupancyFile(sq)];
    return (r2 | r1) ;
}
```

The function above returns all the moves and potential defense of a Rook on square sq.

To have only the valid moves we just need to do something like:

```
public ulong GetRookMoves(int sq,Side color)
{
    ulong enemyempty = (color == Side.Black ? WhitePieces : BlackPieces)|EmptySquares;
    return GetRookMovesAndDefense(sq) & enemyempty;
}
```

As shown, we only need to bitwise end the result with the bitmask of the empty part and part of the opponent color.

The same we do for bishops:

```
public ulong GetBishopMovesAndDefense(int sq)
{
    ulong r1 = diag45moves[sq][Occupancy45(sq)];
    ulong r2 = diag135moves[sq][Occupancy135(sq)];
    return (r2 | r1) ;
}

public ulong GetBishopMoves(int sq, Side color)
{
    ulong enemyempty = (color == Side.Black ? WhitePieces : BlackPieces) |
EmptySquares;
    return GetBishopMovesAndDefense(sq) & enemyempty;
}
```

For the queen just take the bitwise or of the two movement bitmask.

Conclusion.

These approach is faster than the traditional array for acceding the pre-calculated moves for sliding pieces. The speed depends on how fast the dictionary we use is. The standard .NET dictionary class worked very well for me. The counter part is that the program will initialize a big quantity of static data, and we perceive it very slow when it starts, and even the first time we inquiry the part moves. Passed these points the program perform very well. The memory occupation is not so much for the current days personal computer.