Computer Chess (Dec 2019)

Ahmed Ashraf, Ahmed Attia, Ahmed Mahboub, Abdullah Abo-Sedo, Ebrahim Gomaa, *IEEE*

Abstract—**Our problem is that how to obtain the attack set of the next move for sliding pieces (queen, bishop and rock) in computer chess engines.**
But why sliding pieces? The answer is that sliding pieces have so many combinations to be considered.
In this paper we will make a comparison between the classical approach to obtain that and the magic bitboard approach and we will provide the reader with C++ implementation for the magic approach as we aim to provide programmers who write chess engines a discussion of one of the fastest and most versatile move-bitboard generators for individual sliding pieces.

I. Introduction

Computer chess has dozens of areas that need optimizations and improvements especially with the appearance of the AI and Machine Learning which urges scientists to look again in the old problems of the chess engines.

One of the main areas in computer chess and chess engines is how to obtain a move for a piece. There are dozens of algorithms for each piece but in this paper, we will concentrate on the sliding pieces.

Sliding pieces are the bishop, rock, and the queen. They have thousand possibilities as it as they can move any number of moves in many directions on the contrary the pawn who can move only one step.

In the classical approach of obtaining a move for a sliding piece it takes 47 instructions of x86 instructions set but we can do better as we will see that it needs only 15 instructions which is 30% faster than the classical approach.

This approach if called Magic Bitboards which represent the chess matrix as 64-bit number and use something called a magic number to obtain a move.

II. Bitboard Representation and Orientation

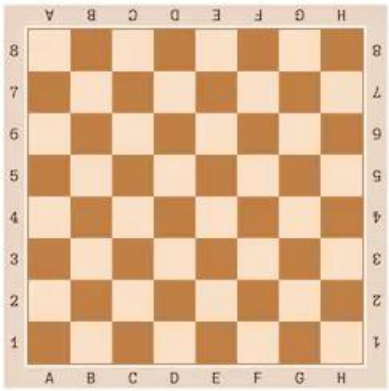
In chess there are 64 squares on the board, therefore 64-bit numbers are required to be used as bitboards as shown

```
/*The following definition of an unsigned 64-bit number that will be used
 *throughout the rest of this paper assumes a C99 compiler conformance.
 *The current Microsoft, Intel, and GNU compilers will be able to compile
 *this code.
 */

typedef unsigned long long U64;

/*The following macro is used to append the appropriate suffix to unsigned
 *64-bit constants.
 */
#define C64(x) x##ULL
#define U64FULL C64(0xFFFFFFFFFFFFFFFF)
```

For this text our bitboard representation has A1 being the least significant bit4, H8 being the most significant bit5, and the squares in between represented by counting up through ranks. To show the representation graphically, here is a chessboard as we see it from white's side:



Here are the respective indices for each bit in the bitboard for the above chessboard.

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

In binary the same chessboard can be represented like this: H8G8F8E8D8C8B8A8H7G7F7E7D7C7B7A7 . . . H2G2F2E2D2C2B2A2H1G1F1E1D1C1B1A1 Although this definition of the bitboard will be used throughout the rest of the text, any particular orientation and geometry of the bitboard may be hashed for move-bitboard generation by magic hashing techniques

III. Methodology

We have two approaches, the classical one and the magic approach.
The classical approach calculates the attack set for the pieces every time we generate a move by generating rays in every attack direction and stop the ray once there is a blocking piece.
On the other hand, the magic approach is a multiply-right-shift perfect hashing algorithm to map the attack set for the pieces to current game state or the blocking state as we preinitialized all attack sets for sliding pieces so we can map the current state to its attack sets.

IV. Classical approach

In this approach, a matrix is generated at the engine starting. in the 8 directions for every square.

RAYS[NORTH_EAST][25]

$$\begin{pmatrix} . & . & . & . & . & 1 & . & . \\ . & . & . & . & 1 & . & . & . \\ . & . & . & 1 & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

RAYS[NORTH][10]

$$\begin{pmatrix} . & . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & 1 & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

RAYS[EAST][4]

$$\begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & 1 & 1 \end{pmatrix}$$

We can write a C++ Code that can obtain these rays first we need a function that can obtain a ray in any direction we called it ‘’MaskBits’’ that tacks index of the cell and direction of the attack.

```
U64 MaskBits(const Direction& direction, const
int index)
  U64 bitboard  =   0ULL;
  int next_index= index;
  while (direction.NextIndex(next_index) >= 0) {
    next_index = direction.NextIndex(next_index);
    bitboard  |=  (1ULL <<  next_index);
  }
  return  bitboard;
}
```

[C++ implementation for obtaining attack Rays]

Then we can initialize 2d array that have all attack rays in the 8 directions for each cell.
We called this array or table ‘’ RAYS’’ and its size 64*8

And here is the initialize function.:

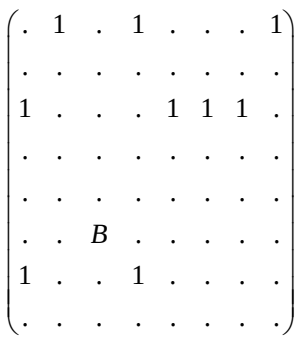
```
Void  initialize_rays() {
  for  (int  i  =  0;  i  <  Direcitions;  i++)
  {
    for  (int  j  =  0;  j  <  64;  j++)
    {
      Direction D=  Direction::NORTH;
      if  (i  ==  0)  D  =  Direction::NORTH;
      else  if  (i  ==  1)  D  =  Direction::SOUTH;
      else  if  (i  ==  2)  D  =  Direction::EAST;
      else  if  (i  ==  3)  D  =  Direction::WEST;
      else if (i == 4) D = Direction::NORTH_EAST;
      else if (i == 5) D = Direction::NORTH_WEST;
      else if (i == 6) D = Direction::SOUTH_EAST;
      else if (i == 7) D = Direction::SOUTH_WEST;
      RAYS[i][j]  =  MaskBits(D,j);
    }}}
```

[C++ implementation for initializing rays]

Rays for bishop and rock are calculated ray by ray for each of the 8 directions. Simply the rays for queen can be calculated by tacking bitwise OR between bishop and rook.

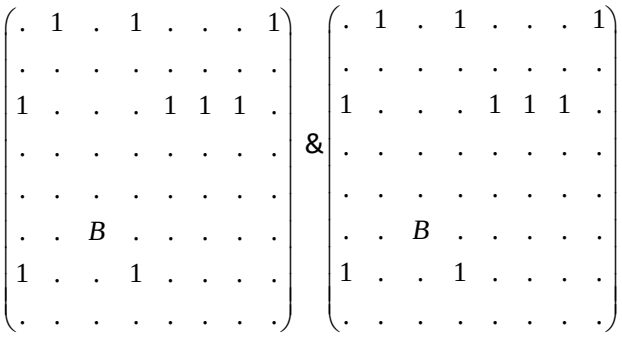
Let B is Bishop in the next example and we want to calculate all moves for B in c3 cell and the blocking pieces just like this:

BLOCKERS

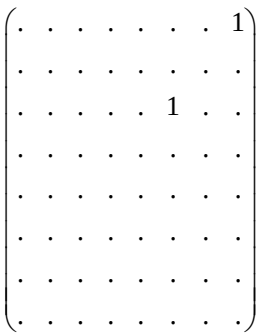


Let's try to obtain the northwest ray for B, if we take bitwise AND for blockers matrix and the northwest matrix for B in cell c3 we get the real blocker that blocks B in this Direction.

BLOCKERS RAYS [northeast][c3]

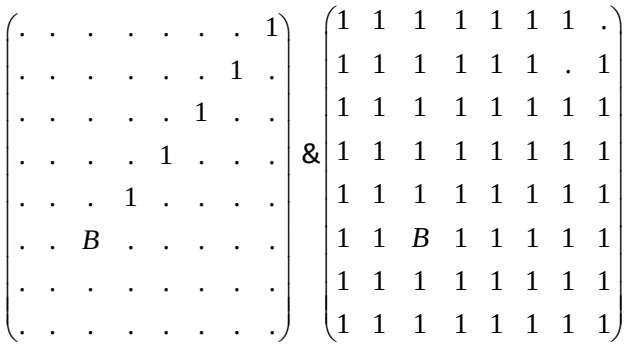


=



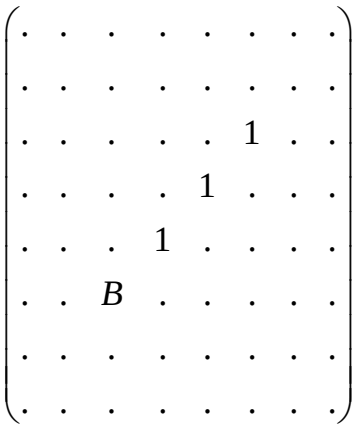
Now we just want to know the position for the first nonzero bit in this Direction in the maskedBlockers Using instruction bsf (bits can forward).

RAYS[NORTH_WEST[c3] ~RAYS[NORTH_WEST[f6]



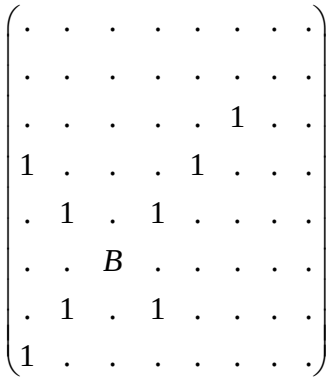
=

result



Similarly, we can obtain the other 3 Directions for B

Attack set



We now ready to implement our classical functions to get attack set for Bishop, Rock and Queen.

C++ implementation to get the Bishop attack:

```
U64 Bishop_classical(int indx, U64 blocker)
{
  U64 attacks = 0ULL;
  attacks |=  RAYS[Direction::NORTH_WEST][indx];
  if (RAYS[Direction::NORTH_WEST][indx] & blocker)
  {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::NORTH_WEST][indx]  &
blocker);
    if(valid){
      attacks &=
~RAYS[Direction::NORTH_WEST][blockerIndex];
    }
  }

attacks |=  RAYS[Direction::NORTH_EAST][indx];
  if (RAYS[Direction::NORTH_EAST][indx] & blocker)
  {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::NORTH_EAST][indx]  &
blocker);
    if(valid){
      attacks &=
~RAYS[Direction::NORTH_EAST][blockerIndex];
    }
  }

attacks |=  RAYS[Direction::SOUTH_EAST][indx];
  if (RAYS[Direction::SOUTH_EAST][indx] & blocker)
  {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::SOUTH_EAST][indx]  &
blocker);
    if(valid){
      attacks &=
~RAYS[Direction::SOUTH_EAST][blockerIndex];
    }
  }

attacks |=  RAYS[Direction::SOUTH_WEST][indx];
  if (RAYS[Direction::SOUTH_WEST][indx] & blocker)
  {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::SOUTH_WEST][indx]  &
blocker);
    if(valid){
      attacks &=
~RAYS[Direction::SOUTH_WEST][blockerIndex];
    }
  } return  attacks;
}
```

C++ implementation to get the Rook attack:

```
U64 Rook_classical(int indx, U64 blocker)
{
  U64 attacks = 0ULL;
  attacks |=  RAYS[Direction::NORTH][indx];
  if (RAYS[Direction::NORTH][indx] & blocker)    {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::NORTH][indx] & blocker);
    if(valid){
      attacks &=
~RAYS[Direction::NORTH][blockerIndex];
    }
  }

attacks |=  RAYS[Direction::EAST][indx];
  if (RAYS[Direction::EAST][indx] & blocker)    {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::EAST][indx]  & blocker);
    if(valid){
      attacks &=
~RAYS[Direction::EAST][blockerIndex];
    }
  }

attacks |=  RAYS[Direction::SOUTH][indx];
  if (RAYS[Direction::SOUTH][indx] & blocker)    {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::SOUTH][indx] &      blocker);
    if(valid){
      attacks &=
~RAYS[Direction::SOUTH][blockerIndex];
    }
  }

attacks |=  RAYS[Direction::WEST][indx];
  if (RAYS[Direction::WEST][indx] & blocker)    {
    unsigned long  blockerIndex;
    unsigned char  valid =
_BitScanForward64(&blockerIndex,
      RAYS[Direction::WEST][indx]  &      blocker);
    if(valid){
      attacks &=
~RAYS[Direction::WEST][blockerIndex];
    }
  } return  attacks;
}
```

C++ implementation to get the Queen attack:

```
U64 Queen_classical(int  indx, U64 blocker)
{
  return  Bishop_classical(indx, blocker)|
Rook_classical(indx, blocker);
}
```

V. Magic Bitboard

Magic Bitboards is not special type of bitboards but the word magic refers to technique to
generate attacks for sliding pieces quickly.
In this approach we just precalculated all possible attack sets for all squares, for all
possible sets of blockers.
So, we need a hash table to map blocking states (that really blocks our piece) to attack
sets. the blocking state can be obtained by taking bitwise AND between the attack mask and all
blockers.

RookMask

$$\begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & 1 & 1 & 1 & 1 & . & 1 & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix} \& \begin{pmatrix} 1 & 1 & . & . & . & 1 & . & . \\ . & 1 & 1 & 1 & 1 & . & . & 1 \\ . & 1 & . & 1 & . & . & . & 1 \\ . & . & . & . & . & . & . & . \\ . & 1 & . & . & . & . & 1 & . \\ . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ 1 & . & . & . & . & . & . & . \end{pmatrix}$$

blockers

=

maskedBlockers

$$\begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & 1 & . & 1 & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

The masked blockers bitboard is what our hash table will map to attack set.
The attack set that the hash table will map this mask for is this following one

$$\begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & 1 & . & 1 & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

Now how to make our table. Maybe you just think that we do not need any hash function at all using maskblockers as a key but this is so much to store as we have 264 possible blockers set and we also have 64 square that for the position of our sliding piece so we have to store array of 64*264 about (147.57exabytes).

So, we need a smaller key for our maskBlockers that we can index our preinitialized attack set table. So, we need a perfect hash function and very quick to execute (fewer instructions).

maskedBlockers

$$\begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & 1 & . & 1 & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

magicNum

$$* \; 0x12000810020004$$

=

multResult

$$\begin{pmatrix} . & . & 1 & . & 1 & . & 1 & . \\ 1 & . & . & . & . & . & . & . \\ . & 1 & . & 1 & . & 1 & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

multResult KEY

$$\begin{pmatrix} . & . & 1 & . & 1 & . & 1 & . \\ 1 & . & . & . & . & . & . & . \\ . & 1 & . & 1 & . & 1 & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & . & . \end{pmatrix} >>54 = \begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ 1 & . & . & . & . & . & . & . \\ . & . & . & . & 1 & . & 1 & . \end{pmatrix}$$

Ok that maybe seems so strange or like just a magic but let us see what happened. the mask blocker if we represent it as a Hex then it will be a large number so we need to make it smaller as it is our key in our hash table so our magic number is just another matrix that can be multiplied to maskedBlockers so when we just represent our magic num in binary we just get 64 bit matrix so our result is that messy matrix ?! actually it is not messy at let see what is there. We will just replace our blockers with A, B, C to keep tracking with them

$$\begin{pmatrix} . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & B & . & C & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & A & . & . \\ . & . & . & . & . & . & . & . \end{pmatrix} *0x12000810020004$$

$$= \begin{pmatrix} . & . & A & . & B & . & C & . \\ 1 & . & . & . & . & . & . & . \\ . & 1 & . & 1 & . & 1 & . & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & . & . \end{pmatrix}$$

So, what happened her that our A, B, C went to the highest bits in the result. As we just need the top 10 (as they are the attack/blocker cells for our B at c3) so we just need to keep them so now we will shift our number 54 bit to keep the top 10 bit in the lowest 10 bit so our number result number is minimum as possible

Using this method, the preinitialized attack tables are about 800 KiB for rooks and 38 KiB for bishops.

A C++ implementation for Bishop:

```
U64 BishopAttacks(const U64 bitboard, const int index) const {
    const Magic& m = bishop_magics_[index];
    return bishop_attack_table_[AttackTableIndex(bitboard, m)];
  }
```

A C++ implementation for Rook:

```
U64 RookAttacks(const U64 bitboard, const int index) const {
    const Magic& m = rook_magics_[index];
    return rook_attack_table_[AttackTableIndex(bitboard, m)];
  }
```

A C++ implementation for Queen:

```
U64 QueenAttacks(const U64 bitboard, const int index) const {
    return RookAttacks(bitboard, index) | BishopAttacks(bitboard, index);
  }
```

VI. Results

Now let's see what is the difference between these two approaches when we run these two methods and comparing the assembly file, we found out that the magic approach tacks less instruction than the classical one around 1/3 of the classical instruction. We also end up with huge improvement of time complicity as the magic approach faster than the classical one by about 30%.

VII. Discussion And conclusion

Computer chess has dozens of areas that need optimizations and improvements especially move generation, the faster we obtained a move the more efficient the chess engine becomes here we disused how to make our move faster by 30% by using magic Bitboard and how to implement it in simple C++ functions that programmers can use to implement their chess engines. all this functions can be found in this gihub repository https://github.com/aashrafh/AlgoBrain.

VIII. Future Work

Now we are Working for 2 different baths to improve chess engines:
1-we look for chess Engines that do not use magic number and apply magic number on them.
2-we want to extend our research to implement AI engine that is more efficient than classical Engines

IX. References

[1] Magic Move-Bitboard Generation in Computer Chess Pradyumna Kannan April 30, 2007.
[2] Maurizio Monge, “On perfect hashing of numbers with sparse digit representation via multiplication by a constant”, 2010
[3] www.chessprogramming.org.
[4] Andrew M. Colman, “Game Theory and its Applications”, 2013
[5] Morton D. Davis, “Game Theory: a nontechnical introduction”, 2012
[6] Werda Buana Putra and Lukman Heryawan, “Applying Alpha-Beta Algorithm In A Chess Engine”, 2017