Journal of Theoretical and Applied Computer Science
ISSN 2299-2634 (printed), 2300-5653 (online)

Vol. 8, No. 4, 2014, pp. 64–71
http://www.jtacs.org

Using bitboards for move generation in chess for three

Jacek Klimaszewski

Faculty of Computer Science and Information Technology, West Pomeranian University of Technology, Szczecin, Poland

jklimaszewski@wi.zut.edu.pl

Abstract: *"Chess for three" is an interesting chess variant, but it is still a bit underrated. This paper describes game rules shortly and proposes bitboard representation, which can be used by a computer chess program in the move generation procedure. Top-rated chess programs (e.g. Stockfish, Rybka) use it, therefore author decided to adapt it in hope of getting similar performance. Array representation is also discussed to show its drawbacks in this case.*

Keywords: *bitboard, chess for three, computer chess*

1. Introduction

The chess game from its beginning has been evolving — many variants appeared, such as chinese chess (*Xiangqi*), Chess960 et caetera [1]. However, for a long time chess was still a two-player game. The first known attempt of involving a third player dates back to the year 1722, when Philip Marinelli published his idea [2]. In the nineties of 20th century Polish variant, based on the idea of Jacek Filek, came into existence [3]. Inclusion of the third player has varied game significantly, causing the gameplay to become more dynamic, because there is only one winner, hence sometimes one player has to help another player to avoid defeat. Even though more than 20 years passed, this chess variant is not so popular yet.

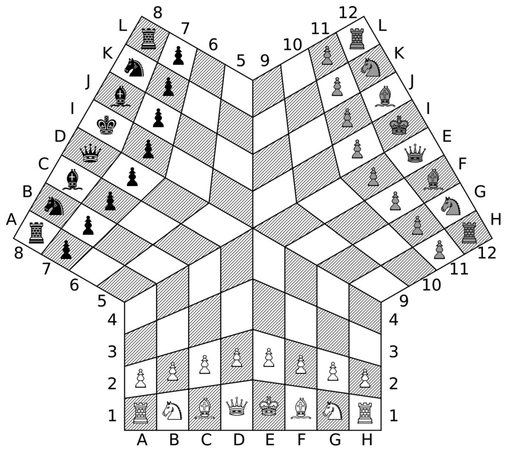


Figure 1. Initial position in *chess for three* (red pieces are marked as grey).

Computer chess program that plays this chess variant could raise its popularity. To make writing of the computer chess program easier, bitboard representation is proposed because of its high performance in the move generation task. With an efficient board representation, people may attempt to write a tree-search procedure and an evaluation function.

The outline of the article is as follows. Section 2 puts forward a game rules of *chess for three*. In the Section 3 array representation is analysed briefly and bitboard representation is described. Section 4 presents results based on a number of visited nodes in the game tree. The article concludes with an outlook on future work.

2. Game rules

The game is played on the board with 96 squares, alternatingly dark and white (Figure 1). Each player has 16 pieces, the same as in the traditional chess, and they move in the same way as standard chess pieces. Untypical board shape makes some of them more mobile (Figure 2, 3, 4, 5, 6, 7).

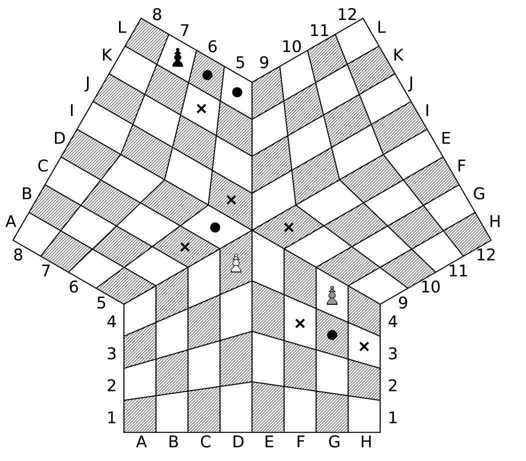


Figure 2. Pawn's moves (× denotes capture).

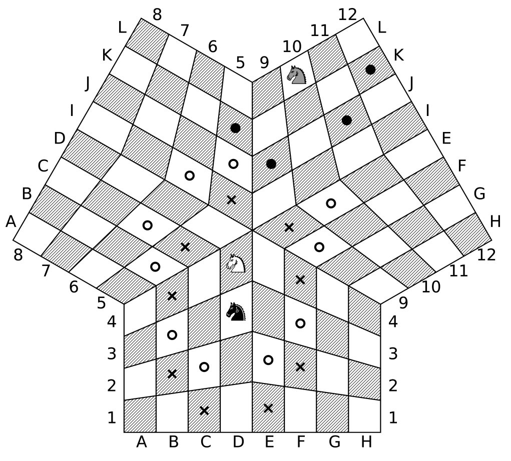


Figure 3. Knight's moves (× — black knight).

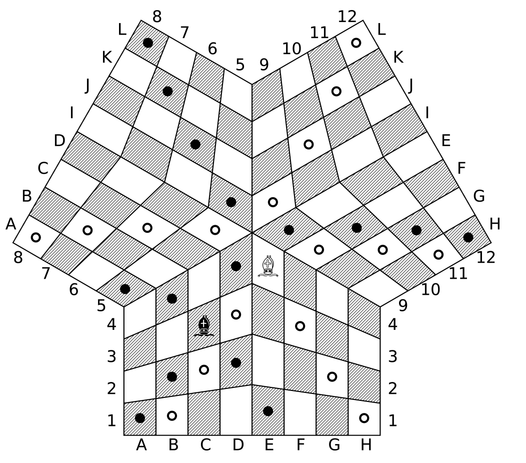


Figure 4. Bishop's moves.

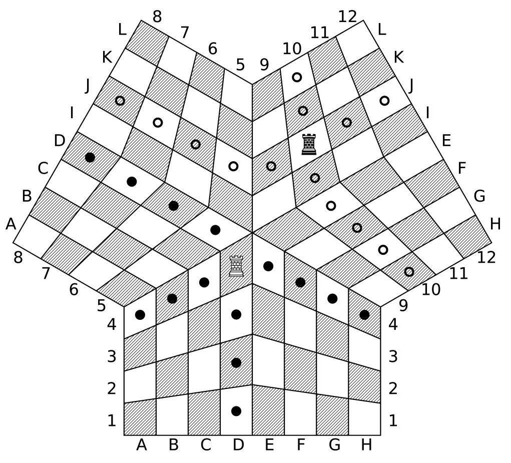


Figure 5. Rook's moves.

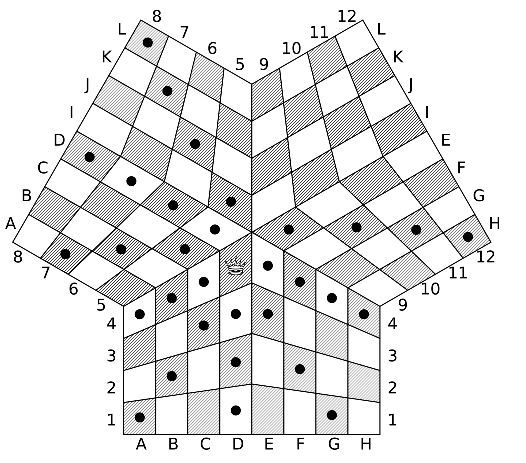


Figure 6. Queen's moves.

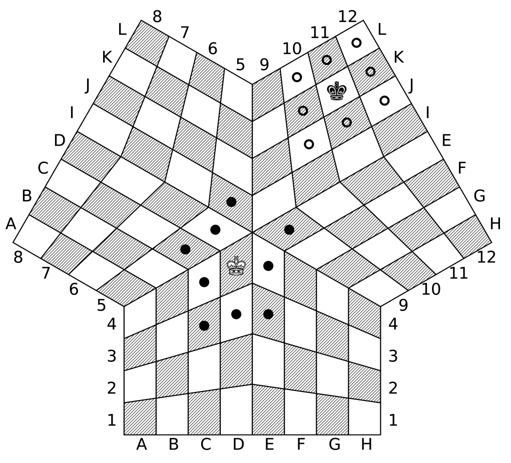


Figure 7. King's moves.

Before the game starts, red player may exchange positions of a red king and a red queen. Red makes the first move, the next move is made by white, then black makes the next move and so forth. The game lasts until one of the players is checkmated or a draw occurs (by stalemate, by repetition and so on).

Other rules (castling, pawn promotion) are the same with exception of *en passant*, which does not exist in this chess variant. More information about the game may be found in [3].

3. Bitboard representation in *chess for three*

Untypical shape of the chessboard hampers usage of an array as a storage of the position. Everyone may notice, that it is not obvious how to map squares to indices — in case of traditional chessboard it is far more intuitive; also some improved array representations exists for it [4]. Exemplary projection, which is used later, is depicted on the Figure 8. It is some compromise between ease of move generation for one side and complication for the other sides. Of course, other projections exist that have different advantages and disadvantages.

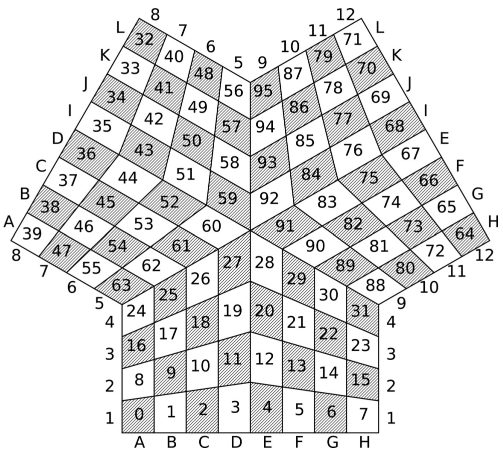


Figure 8. Square-to-index projection.

Because of this difficulty, array representation was not considered anymore and work focused on finding more suitable bitboard representation. The idea of using bitmaps to represent positional information is not new [5]. Probably the first chess program that used them was *Kaissa*, written by Russian programmers, which won the first *World Computer Chess Championship* in Stockholm [6].

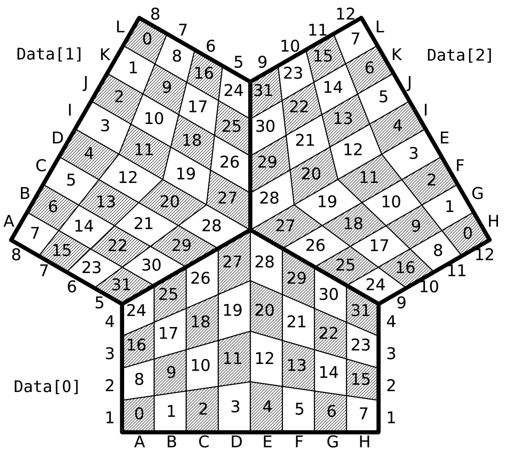


Figure 9. Square-to-bit projection.

As can be seen, chessboard contains 96 squares, so 96 bits are needed. None of the predefined types in C++ has the size of 96 bits. To overcome this problem, three variables of type *unsigned int* can be used, but a much better idea is to use an array of 3 integers [7]. The projection of the bits to the squares is shown on the Figure 9. To simplify operations on this data type, it has been wrapped in the class and useful operators have been overloaded:

Listing 1. Bitboard class declaration.

```
typedef unsigned int uint;
class Bitboard
{
private:
    uint Data[3];
public:
    Bitboard(uint Data0 = 0, uint Data1 = 0, uint Data2 = 0);
    bool operator ==(const Bitboard& n) const;
    bool operator !=(const Bitboard& n) const;
    operator uint() const;
    Bitboard operator &(const Bitboard& n) const;
    Bitboard operator |(const Bitboard& n) const;
    Bitboard operator ^(const Bitboard& n) const;
    Bitboard operator ~() const;
    int LS1B() const;
    bool IsBitSet(uint n) const;
    void SetBit(uint n);
    void ClearBit(uint n);
    uint GetData(uint n) const;
    Bitboard AdvanceWhitePawns() const;
    Bitboard AdvanceBlackPawns() const;
    Bitboard AdvanceRedPawns  () const;
    Bitboard WhitePawnsAttacks() const;
    Bitboard BlackPawnsAttacks() const;
    Bitboard RedPawnsAttacks  () const;
};
```

Below there is a source code of a few methods. Other methods are very similar, so they are not listed here.

Listing 2. Source code of some methods.

```
Bitboard::operator uint() const
{
    return (Data[0] | Data[1] | Data[2]);
}
Bitboard Bitboard::operator &(const Bitboard& n) const
{
    return Bitboard(Data[0] & n.Data[0], Data[1] & n.Data[1], Data[2] & n.Data[2]);
}
uint Bitboard::GetData(uint n) const
{
    return (n < 96 ? Data[n >> 5] >> (n & 31) : 0);
}
void Bitboard::ClearBit(uint n)
{
    if (n < 96)
        Data[n >> 5] &= ~(1 << (n & 31));
}
Bitboard Bitboard::AdvanceWhitePawns() const
{
    uint n = (Data[0] >> 24);
    n = ((n & 0x55) << 1) | ((n >> 1) & 0x55);
    n = ((n & 0x33) << 2) | ((n >> 2) & 0x33);
    n = ((n & 0x0f) << 4) | ((n >> 4) & 0x0f);
    return Bitboard(Data[0] << 8,
                    (Data[1] >> 8) | ((n & 0xf0) << 24),
                    (Data[2] >> 8) | ((n & 0x0f) << 24));
}
Bitboard Bitboard::WhitePawnsAttacks() const
{
    return Bitboard(((Data[0] & 0xfefefefe) << 7) | ((Data[0] & 0x7f7f7f7f) << 9),
                    ((Data[1] & 0x7f7f7f7f) >> 7) | ((Data[1] & 0xfefefefe) >> 9),
                    ((Data[2] & 0x7f7f7f7f) >> 7) | ((Data[2] & 0xfefefefe) >> 9));
}
```

Casting operator is helpful to check whether bitboard is empty (equal to 0) or not. `GetData` method has almost the same behaviour as a right shift operator. `AdvanceWhitePawns` method moves bits in the same way as white pawn steps forward. `WhitePawnsAttacks` method is used to generate pawn's captures. In the actual implementation, `LS1B` method utilises assembly instruction `bsf` (*Bit Scan Forward*) to extract position of the *least significant 1 bit*. The same result may be obtained using *de Bruijn sequences* [8].

Listing 3. Source code of a LS1B method that uses *de Bruijn sequences.*

```
const unsigned int debruijn32 = 0x4653adf, index32[] = {0, 1, 2, 6, 3, 11, 7, 16, 4, 14, ↪
    ↩12, 21, 8, 23, 17, 26, 31, 5, 10, 15, 13, 20, 22, 25, 30, 9, 19, 24, 29, 18, 28, 27};
int Bitboard::LS1B() const
{
    if (Data[0] != 0) return index32[((Data[0] & -Data[0]) * debruijn32) >> 27];
    if (Data[1] != 0) return index32[((Data[1] & -Data[1]) * debruijn32) >> 27] + 32;
    if (Data[2] != 0) return index32[((Data[2] & -Data[2]) * debruijn32) >> 27] + 64;
    return -1;
}
```

To store whole position, each bitboard for each piece's type must be created, so 18 bitboards are needed. As can be noted, position of the king does not have to be put into the bitboard, because each player always has one and only one king. To facilitate some operations, it is better to keep those bitboards in the array (e.g. `Bitboard Pawns[3], Knights[3]` and so on). Also creation of another bitboards (representing occupancy of all pieces and occupancy of all pieces of each player) will save time in move generation process. All of those auxiliary bitmaps must be updated every move.

At last, it is a good idea to use an array representation in conjunction with a bitboard representation, because it will be easier to determine whether capture occurs and what piece is captured (if any) [4].

3.1. Move generation of non-sliding pieces

For non-sliding pieces (i.e. pawn, knight and king) move generation is relatively easy. If pawn stays on the "central edge" (4th rank for white, 5th for black, 9th for red), auxiliary bitboards representing attacks from those squares (e.g. attacks of white pawn from Figure 2 are given by `Bitboard(0x00000000u, 0x28000000u, 0x08000000u)`) are used.

To generate knight's possible moves (king is handled similarly), precomputed array of 96 bitboards is used, where every bitboard contains available moves from a given square. Figure 10 depicts bitboard representing available moves of the knight from the D4 square. Below there is a part of the knight's move generation procedure.

```
copy = Knights[Side];
while ((from = copy.LS1B()) != -1)
{
    copy.ClearBit(from);
    attacks = KnightAttacks[from] & ~Pieces[Side];
    while ((to = attacks.LS1B()) != -1)
    {
        attacks.ClearBit(to);
        add_move(from, to);
    }
}
```

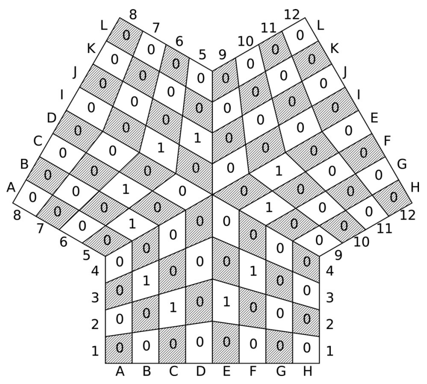
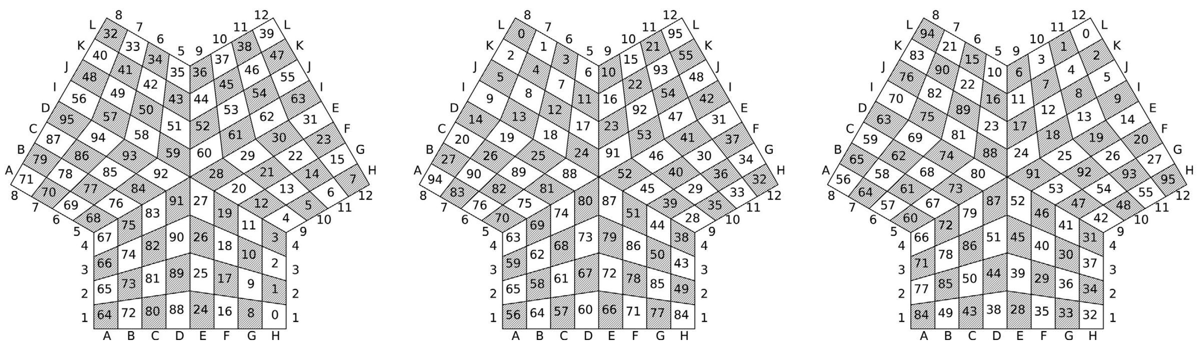


Figure 10. Bitboard representing attacks of a knight from D4 square.

3.2. Move generation of sliding pieces

Move generation of sliding pieces is handled in a different way, because sliding pieces stop when they encounter a piece of any kind. For detecting possible moves in the rank (row), two dimensional pre-calculated array of bitboards `RankAttacks[96][256]` is needed, where the first index denotes square and the second index stands for occupancy of the rank [9]. As Hyatt pointed out [10], outer squares do not change attack, so they may be discarded — it reduces size of the array 4 times.



(a) Files. (b) Left diagonals. (c) Right diagonals.

Figure 11. Projection of files and diagonals.

The same idea can be used to generate possible moves in the file (column), but earlier files have to be mapped, because squares in the file are not adjacent in the bitboard. This projection is known as a rotation by 90 degree [10] and it is shown in the Figure 11.

Now it is possible to generate moves of a rook. Below there is a part of the source code, which does this task. Actually, one dimensional array `RankAttacks[96*64]` was used.

```
copy = Rooks[Side];
while ((from = copy.LS1B()) != -1)
{
    copy.ClearBit(from);
    attacks = RankAttacks[(from << 6) | (Occupied.GetData(RankShift[from]) & 0x3f)];
    attacks |= FileAttacks[(from << 6) | (Occupied_R90.GetData(FileShift[from]) & 0x3f)];
    attacks &= ~Pieces[Side];
    while ((to = attacks.LS1B()) != -1)
    {
        attacks.ClearBit(to);
        add_move(from, to);
    }
}
```

Auxiliary arrays `RankShift` and `FileShift` are given below.

```
const int RankShift[96] =
{
     1,  1,  1,  1,  1,  1,  1,  1,
     9,  9,  9,  9,  9,  9,  9,  9,
    17, 17, 17, 17, 17, 17, 17, 17,
    25, 25, 25, 25, 25, 25, 25, 25,
    33, 33, 33, 33, 33, 33, 33, 33,
    41, 41, 41, 41, 41, 41, 41, 41,
    49, 49, 49, 49, 49, 49, 49, 49,
    57, 57, 57, 57, 57, 57, 57, 57,
    65, 65, 65, 65, 65, 65, 65, 65,
    73, 73, 73, 73, 73, 73, 73, 73,
    81, 81, 81, 81, 81, 81, 81, 81,
    89, 89, 89, 89, 89, 89, 89, 89
};
```

```
const int FileShift[96] =
{
    65, 73, 81, 89, 25, 17,  9,  1,
    65, 73, 81, 89, 25, 17,  9,  1,
    65, 73, 81, 89, 25, 17,  9,  1,
    65, 73, 81, 89, 25, 17,  9,  1,
    33, 41, 49, 57, 89, 81, 73, 65,
    33, 41, 49, 57, 89, 81, 73, 65,
    33, 41, 49, 57, 89, 81, 73, 65,
    33, 41, 49, 57, 89, 81, 73, 65,
     1,  9, 17, 25, 57, 49, 41, 33,
     1,  9, 17, 25, 57, 49, 41, 33,
     1,  9, 17, 25, 57, 49, 41, 33,
     1,  9, 17, 25, 57, 49, 41, 33
};
```

Move generation in diagonals is handled similarly. To save memory, main diagonals are processed separately, so there are 4 arrays instead of 2: `LeftDiagAttacks[96][32]` and `LeftMainDiagAttacks[12][512]` for the left diagonals and respectively there are `RightDiagAttacks[96][32]` and `RightMainDiagAttacks[12][512]` for the right diagonals. Moreover, diagonals do not have fixed size, hence 2 more arrays containing masks are needed. Those auxiliary arrays are listed below.

```
const int R45_Shift[96] =
{
    85, 50, 44, 39, 29, 36, 34, 33,
    78, 85, 50, 44, 39, 29, 36, 34,
    72, 78, 85, 50, 44, 39, 29, 36,
    67, 72, 78, 85, 50, 44, 39, 29,
    85, 78, 72, 67, 61, 58, 65, 57,
    22, 85, 78, 72, 67, 61, 58, 65,
    16, 22, 85, 78, 72, 67, 61, 58,
    11, 16, 22, 85, 78, 72, 67, 61,
    85, 22, 16, 11,  7,  4,  2,  1,
    50, 85, 22, 16, 11,  7,  4,  2,
    44, 50, 85, 22, 16, 11,  7,  4,
    39, 44, 50, 85, 22, 16, 11,  7
};
```

```
const int L45_Shift[96] =
{
    57, 65, 58, 61, 67, 72, 78, 85,
    65, 58, 61, 67, 72, 78, 85, 50,
    58, 61, 67, 72, 78, 85, 50, 44,
    61, 67, 72, 78, 85, 50, 44, 39,
     1,  2,  4,  7, 11, 16, 22, 85,
     2,  4,  7, 11, 16, 22, 85, 78,
     4,  7, 11, 16, 22, 85, 78, 72,
     7, 11, 16, 22, 85, 78, 72, 67,
    33, 34, 36, 29, 39, 44, 50, 85,
    34, 36, 29, 39, 44, 50, 85, 22,
    36, 29, 39, 44, 50, 85, 22, 16,
    29, 39, 44, 50, 85, 22, 16, 11
};
```

```
const int R45_Mask[96] =
{
511,  31,  15,   7,   3,   1,   0,   0,
 31, 511,  31,  15,   7,   3,   1,   0,
 15,  31, 511,  31,  15,   7,   3,   1,
  7,  15,  31, 511,  31,  15,   7,   3,
511,  31,  15,   7,   3,   1,   0,   0,
 31, 511,  31,  15,   7,   3,   1,   0,
 15,  31, 511,  31,  15,   7,   3,   1,
  7,  15,  31, 511,  31,  15,   7,   3,
511,  31,  15,   7,   3,   1,   0,   0,
 31, 511,  31,  15,   7,   3,   1,   0,
 15,  31, 511,  31,  15,   7,   3,   1,
  7,  15,  31, 511,  31,  15,   7,   3
};
```

```
const int L45_Mask[96] =
{
  0,   0,   1,   3,   7,  15,  31, 511,
  0,   1,   3,   7,  15,  31, 511,  31,
  1,   3,   7,  15,  31, 511,  31,  15,
  3,   7,  15,  31, 511,  31,  15,   7,
  0,   0,   1,   3,   7,  15,  31, 511,
  0,   1,   3,   7,  15,  31, 511,  31,
  1,   3,   7,  15,  31, 511,  31,  15,
  3,   7,  15,  31, 511,  31,  15,   7,
  0,   0,   1,   3,   7,  15,  31, 511,
  0,   1,   3,   7,  15,  31, 511,  31,
  1,   3,   7,  15,  31, 511,  31,  15,
  3,   7,  15,  31, 511,  31,  15,   7
};
```

Now it is possible to generate moves of a bishop. Below there is a source code, which does this job. As mentioned earlier, one dimensional arrays of precomputed bitmaps were used.

```
copy = Bishops[Side];
while ((from = copy.LS1B()) != -1)
{
    copy.ClearBit(from);
    if (Mask[from] & RightMainDiagMask)
        attacks = RightMainDiagAttacks[((Rot45Right[from] - R45_Shift[from] + 1) << 9) | (Occupied_R45.GetData(R45_Shift[from]) & R45_Mask[from])];
    else
        attacks = RightDiagAttacks[(from << 5) | (Occupied_R45.GetData(R45_Shift[from]) & R45_Mask[from])];
    if (Mask[from] & LeftMainDiagMask)
        attacks |= LeftMainDiagAttacks[((Rot45Left[from] - L45_Shift[from] + 1) << 9) | (Occupied_L45.GetData(L45_Shift[from]) & L45_Mask[from])];
    else
        attacks |= LeftDiagAttacks[(from << 5) | (Occupied_L45.GetData(L45_Shift[from]) & L45_Mask[from])];
    attacks &= ~Pieces[Side];
    while ((to = attacks.LS1B()) != -1)
    {
        attacks.ClearBit(to);
        add_move(from, to);
    }
}
```

Arrays *Rot45Left* and *Rot45Right* represent projections of the diagonals, which are given on the Figure 11.

4. Experimental results

Board representation and move generator were written in C++ and built using g++ 4.8.2 with parameter -O2 in Ubuntu 14.04 x64 OS running on the Samsung NP-RC520-S06PL notebook. Then program was traversing the same game tree 10 times using depth-first search algorithm with depth limited to 6 levels and it was counting visited nodes. Starting from the initial position, the average time was 22.9348 seconds to visit all 87,828,597 nodes, which gives average speed of $\approx$ 3,829,490 nodes per second. When using *de Bruijn sequences* to index LS1B instead of inline assembly, the average time was $\approx$ 23.276 seconds, which gives average speed of $\approx$ 3,773,354 nodes per second.

Both results are promising. Surprisingly the difference is relatively small. It should be noted that eventual speed will be lower because of calls to evaluation function in the tree-search procedure.

5. Conclusions and future research

In this paper it was explained how bitboards can be used in the move generation procedure. It opens new doors for writing a computer program that could play this chess variant, because board representation is one of the three components of a chess program — the other two are tree-search procedure and evaluation function.

Probably the most challenging part of writing a computer chess program will be tree-search procedure. In the literature there are a few algorithms (e.g. $\max^n$ [11] with some modifications, best-reply search [12], paranoid search [13]), but maybe some other approach should be used.

References

[1] Pritchard, D.: Popular Chess Variants. Batsford Chess Books. B.T. Batsford, 2000. ISBN 9780713485783.

[2] Marinelli, F.: Triple chess. A.J. Valpy, 1826.

[3] Trząski, W.: Chess for three (in Polish). Wydawnictwo Profesjonalnej Szkoły Biznesu, 2001. ISBN 8372300526.

[4] Hyatt, R.: Chess program board representations. *https://cis.uab.edu/hyatt/boardrep.html*. Accessed: 2015-06-15.

[5] Adelson-Velskii, G., Arlazarov, V., Bitman, A., Zhivotovskii, A., Uskov, A.: Programming a computer to play chess. Russian Mathematical Surveys, 25, pp. 221–262, 1970.

[6] Frey, P.: Chess Skill in Man and Machine. Springer New York, 2012. ISBN 9781461255154.

[7] Grimbergen, R.: Using Bitboards for Move Generation in Shogi. ICGA Journal, 30(1), pp. 25–34, 2007.

[8] Leiserson, C., Prokop, H., Randall, K.: Using de Bruijn Sequences to Index a 1 in a Computer Word, 1998.

[9] Heinz, E.: How DarkThought Plays Chess. ICCA Journal, 20(3), pp. 166–176, 1997.

[10] Hyatt, R.: Rotated bitmaps, a new twist on an old idea. ICCA Journal, 22(4), pp. 213–222, 1999.

[11] Luckhartd, C., Irani, K.: An Algorithmic Solution of N-Person Games. In: AAAI'86, pp. 158–162. 1986.

[12] Schadd, M., Winands, M.: Best-Reply Search for Multiplayer Games. IEEE Transactions on Computational Intelligence and AI in Games, 3(1), pp. 57–66, 2011.

[13] Sturtevant, N., Korf, R.: On Pruning Techniques for Multi-Player Games. In: AAAI/IAAI'00, pp. 201–207. 2000.