Using Bitboards for Move Generation in Shogi

Reijer Grimbergen
Department of Informatics, Yamagata University
Jonan 4-3-16, Yonezawa-shi, 992-8510 Japan
E-mail: grim@yz.yamagata-u.ac.jp

Abstract

In this paper it will be explained how to use bitboards for move generation in shogi. In chess, bitboards have been used in most strong programs because of the easy representation of a chess board by a single 64-bit integer. For shogi, a less efficient representation has to be used because a shogi board has 81 squares instead of 64. A representation with an array of three integers is proposed, where each integer represents 27 squares of the board. This representation is then used for move generation in a similar way to the methods used in chess, for example by using rotated bitboards for generating the moves of the sliding pieces rook, bishop and lance. A comparison of move generation speed between the method using bitboards and by using the more common method of attack tables showed that by using bitboards the move generation speed of the program SPEAR was improved by 48.8%.

Keywords: Bitboards, move generation, computer shogi.

1 Introduction

Bitboards are a binary representation of knowledge for all squares on the board for a certain game. The presence of certain information for a square is represented by setting the bit for this square on the board to 1, while the absence is represented by a 0. Bitboards have been used extensively in chess programs because the size of the board ($8 \times 8 = 64$ squares) makes it possible to fit one bitboard into a single 64-bit integer. This makes the representation of bitboards very efficient (no unused memory) and representing new knowledge by combining the information from different bitboards very fast, because logical operations like AND, OR, NOT and XOR can be used.

Shogi has a different board size ($9 \times 9 = 81$ squares), so the convenient single integer representation for one bitboard cannot be used. Despite this, the bitboard representation offers many ways to efficiently represent information used in the search and evaluation part of a game program. This may outweigh the disadvantage of a less than optimal board representation.

Until recently, most shogi programs used attack tables (called *kiki* in Japanese) to represent important search and evaluation knowledge. My program SPEAR also had attack tables, but after reading a discussion on Hiroshi Yamashita's bulletin board [4] about bitboards in August 2005 I decided to further investigate the use of bitboards in shogi. Yamashita's conclusion was that bitboards may not make much difference, but when the program BONANZA, one of the few shogi programs using the bitboard representation won the World Computer Shogi Championships last May, it seemed clear to me that bitboards in shogi deserved more attention.

Yamashita's explanation was very important to understand the difference between using bitboards in chess and shogi and deserves a more prominent place than being buried deeply into the archives of the BBS. However, even with the original paper by Robert Hyatt (author of the strong chess program CRAFTY) [2] and the thorough description on Wikipedia [3] on bitboards, it takes some time to grasp the full idea. Furthermore, there are some details of the representation as given by Yamashita that can be improved. Also, after the implementation was finished, initial tests seemed to show a significant improvement in performance, even though Yamashita had reported only an increase in speed of about 20% for a tsume shogi solver. Finally, by using bitboards, SPEAR became a much more transparent program and there seem plenty of possibilities to use bitboards in different parts of the program in the future.

Therefore, I think that trying the bitboard representation is important for every shogi program, although it will depend on the structure of the individual program whether or not a significant increase in performance can be achieved. By writing this paper, I hope I will make it easier for other

shogi programmers to implement bitboards by explaining how bitboards are used in chess and how this representation was changed to use bitboards in shogi.

In Section 2, the differences between the bitboard representation used in chess programs and a bitboard representation for shogi will be explained, as well as how bitboards can be used to do move generation in a shogi program. In Section 3, move generation speed using bitboards is compared with move generation speed using attack tables. The conclusion is that bitboards also have potential in shogi, but that it depends on the structure of the evaluation function if they will give a significant performance improvement or not (Section 4).

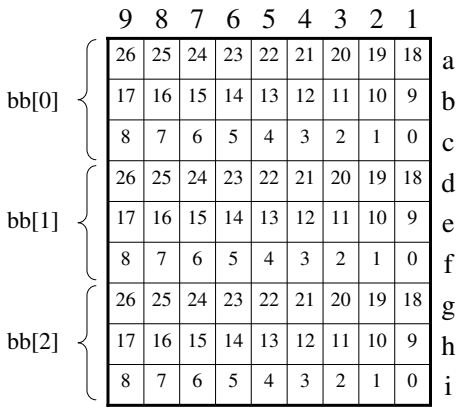

Figure 1: Correspondence between shogiboard squares and bitboard integer bits.

2 Bitboards in Chess and Shogi

As explained in Section 1, bitboards are used to efficiently represent knowledge about a game position. The chess program Crafty uses bitboards for representing many different types of knowledge. For example, where each of the black and white pieces are and where they can move to. There are also bitboards used for evaluation, but here I will explain how bitboards can be used in shogi to do move generation. When this basic use of bitboards is understood, it will not be difficult to use bitboards in other parts of the program like in the evaluation function.

2.1 Basic Bitboards

The most important difference between using bitboards in chess and using bitboards in shogi is that the chessboard has 64 squares, which means that the information about a position fits into a single 64 bit integer. In shogi, this information has to have 81 bits, which will not fit into a single integer. Like Yamashita [4], I use three integers for representing the 81 bits of the board. Yamashita defined a structure of three integers, but as Hoki pointed out [1], it is better to define an array of length 3 (why this is better will become clear later):

```
typedef struct {
   unsigned int bb[3];
} BITBOARD;
```

Because of the different board sizes of chess and shogi, for each bitboard there will 15 bits that are obsolete (3×32 bits-81 bits $= 15$ bits). As will be explained later, especially for piece attacks a lot of pre-calculated bitboards are necessary. Running shogi programs on small computers (for example hand held devices) might be a problem when using bitboards.

Each of the three integers represents one third of the board. In my representation, the first 27 bits of the first integer represent squares 1c to 9a, i.e. the white promotion zone. The first 27 bits of the second integer represent the squares 1f to 9d, the neutral zone in the middle. Finally, the first 27 bits of the third integer represent the squares 1i to 9g, i.e. the black promotion zone (see Figure 1). This representation is different from the initial implementation described by Yamashita, which had the first two integers represent squares 1i to 1b and used the 17 bits of the third integer for squares 2b to 9a. Using 27 bits of each integer is an improvement which was again pointed out by Hoki [1]. The reason for this improvement will become clear after the explanation of the attack bitboards below.

To manipulate bitboards in chess, straightforward logical operations can be used. However, when a bitboard consists of three different integers, this is no longer possible. Logical operations like taking the AND of two bitboards have to be defined as follows:

```
void AndBB(BITBOARD *to,
      BITBOARD *from1, BITBOARD *from2) {
   to->bb[0] = from1->bb[0] & from2->bb[0];
   to->bb[1] = from1->bb[1] & from2->bb[1];
   to->bb[2] = from1->bb[2] & from2->bb[2];
}
```

In my implementation, I use similar operations for `OR`, `XOR` and `NOT`. Furthermore, I use the basic operations `ClearBB` (setting all bits to 0), `SetBB` (setting all bits to 1) and `CopyBB` (copying a bitboard).

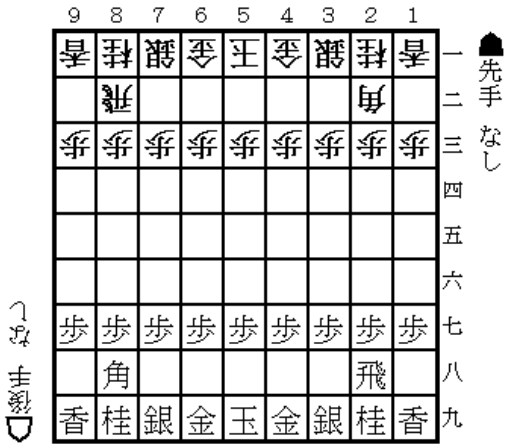

Figure 2: Shogi starting position.

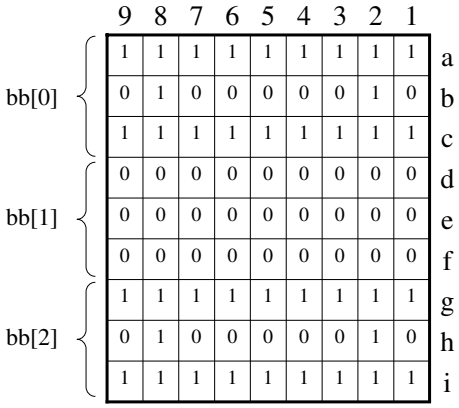

Figure 3: Bit settings for the starting position.

With the basic bitboard data structure defined like this, it is now possible to represent the knowledge needed in a game program using bitboards. For example, useful information is to know which squares are occupied. For the starting position of shogi (Figure 2), the bitboard `Occupied` has 1s for the squares with a black or white piece and 0s for squares without pieces (Figure 3). In the `BITBOARD` data structure this would be:

```
bb[0]: 00000111111111010000010111111111
bb[1]: 00000000000000000000000000000000
bb[2]: 00000111111111010000010111111111
```

In the same way a bitboard `SentePieces` can be used that has 1s for the squares with black pieces and 0s for the other squares. For the starting position of Figure 2, this bitboard would look like this:

```
bb[0]: 00000000000000000000000000000000
bb[1]: 00000000000000000000000000000000
bb[2]: 00000111111111010000010111111111
```

My current implementation has bitboards for the occupied squares (`Occupied`), the black pieces (`SentePieces`), the white pieces (`GotePieces`) and for each of the black and white pieces separately. For example, the bitboard `SenteGold` has 1s for the squares with black golds and for the starting position this bitboard looks as follows:

```
bb[0]: 00000000000000000000000000000000
bb[1]: 00000000000000000000000000000000
bb[2]: 00000000000000000000000000101000
```

These bitboards for position information must be updated each time the position changes, i.e. each time either player makes a move.

2.2 Attack Bitboards

With the bitboard data structure and the logical manipulation of the bits in the bitboard in place, it is now possible to find the squares attacked by each square efficiently using bitboards. This will enable move generation using bitboards instead of attack tables. There is a difference between finding the attacking squares of sliding pieces (rook, bishop, lance) and the other pieces. For non-sliding pieces, the representation of attacking squares with bitboards is simple, but for sliding pieces, the calculation is more complex.

2.2.1 Non-sliding piece attacks

Calculating the squares a non-sliding piece attacks is straightforward. For example, to know where a gold can move to, it is sufficient to pre-calculate the gold attacks for a black or white gold on each of the 81 squares (a total of 162 bitboards). For example, the attack bitboard for a black gold on 5e looks like Figure 4. When the squares that a gold on 5e attack are needed, this bitboard can be accessed by indexing the bitboard array with the number of the square.

2.2.2 Horizontal attacks

The simple pre-calculation method does not work for the sliding pieces rook, bishop and lance because where these pieces can move to depends on how they are blocked by their own pieces or by the pieces of the opponent. For example, in the starting position of shogi, the rook on 2h cannot move forward because there is a pawn on 2g. However, if the pawn on 2g moves to 2f the rook can move to 2g.

	9	8	7	6	5	4	3	2	1	
bb[0]	0	0	0	0	0	0	0	0	0	a
	0	0	0	0	0	0	0	0	0	b
	0	0	0	0	0	0	0	0	0	c
bb[1]	0	0	0	1	1	1	0	0	0	d
	0	0	0	1	0	1	0	0	0	e
	0	0	0	0	1	0	0	0	0	f
bb[2]	0	0	0	0	0	0	0	0	0	g
	0	0	0	0	0	0	0	0	0	h
	0	0	0	0	0	0	0	0	0	i

Figure 4: Attack bitboard for a gold on 5e.

Therefore, the possible movement of a sliding piece depends on the position of the blocking pieces.

The solution to this problem, as pointed out by Hyatt [2] (among others) is to pre-calculate bitboards for all the blocking possibilities. Let's look at the most simple case first, which is the horizontal movement of the rook. If a rook is placed on 7h (like for example in Figure 5), calculate an attack bitboard for the cases in which there is a piece on 9h, 8h, 6h, 5h, 4h, 3h, 2h, 1h or not. For a rank there is a total of $2^9 = 512$ bitboards. The horizontal attack patterns for each of the possible 81 squares is the defined as follows:

```
BITBOARD RankAttacks[81][512];
```

For the example position in the top part of Figure 5, the block pattern on the rank of the rook is 001010110 = 86. In this case the squares that the rook attacks horizontally are 00000000000000000000000000 for the top three ranks (i.e. `bb[0]`), 00000000000000000000000000 for the three ranks in the middle (i.e. `bb[1]`) and 00000000011011000000000000 = 221184 for the bottom three ranks (i.e. `bb[2]`). Therefore, at the start of the program, this pre-calculated value can be added for a rook on 7h (square 66 if 9a is represented by 0 and 1i is represented by 80):

```
RankAttacks[66][86].bb[0] = 0;
RankAttacks[66][86].bb[1] = 0;
RankAttacks[66][86].bb[2] = 221184;
```

Similar, for the block pattern 101011110 = 350 in the bottom part of Figure 5, the squares that the rook attacks horizontally are also 00000000011011000000000000 = 221184, the same as before. This information can then be added to the pre-calculated bitboard:

Figure 5: Attack patterns for black rook on 7h.

```
RankAttacks[66][350].bb[0] = 0;
RankAttacks[66][350].bb[1] = 0;
RankAttacks[66][350].bb[2] = 221184;
```

In this case, for the attacks it does not matter if there is a lance on 9h or not or if there is a pawn on 4h or not. For the pawn on 4h, the block patterns has to be added, because in an actual position, it is not known beforehand if the gold on 5h will be there or not. However, for the lance on 9h, the fact that the horizontal attack does not change is significant: for every block pattern, the horizontal attack value is the same with or without a piece on the edge squares (9h and 1h in the example). This observation made by Hyatt is important, because the array `RankAttacks` already takes up a big chunk of memory: $81 \times 512 \times 3 \times 4$ bytes = 497664 bytes. If the edge squares do not change the attack, then the edge squares can be excluded from the blocking pattern bitboards, leaving $2^7 = 128$ bitboards for each of the 81 squares of the board. The size of the array `RankAttacks` is now reduced to $81 \times 128 \times 3 \times 4$ bytes = 124416 bytes. This is still not small (especially compared to chess, where this bitboard array is three times smaller), but does not require special hardware requirements, even if a number of these arrays are needed to cover vertical attacks and diagonal attacks, as will be explained below.

With these pre-calculated bitboards, it is now easy to get the horizontal attacks for a rook on a square:

```
void CalcRankAtt(BITBOARD *bb, int from) {
   // Get the block pattern from
   // the position bitboard
   int pcs = GetRankBits(&Occupied, from);
   // Strip the bits for the edge squares
   pcs = (pcs & CLEAR9MASK) >> 1;
   // Get the horizontal attacks from the
   // pre-calculated bitboards
   CopyBB(bb, &RankAttacks[from][pcs]);
}
```

```
int GetRankBits(BITBOARD *bb, int sq) {
   // Take the rank bits from bitboard
   // integer 0, 1, or 2
   return (bb->bb[sq_to_bb_index[sq]]
          >> rank_shift_no[sq])
          & ALLONES_9;
}

int sq_to_bb_index[] = {
   0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1,
   2, 2, 2, 2, 2, 2, 2, 2, 2,
   2, 2, 2, 2, 2, 2, 2, 2, 2,
   2, 2, 2, 2, 2, 2, 2, 2, 2,
};

int rank_shift_no[] = {
  18, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   0,  0,  0,  0,  0,  0,  0,  0,  0,
  18, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   0,  0,  0,  0,  0,  0,  0,  0,  0,
  18, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   0,  0,  0,  0,  0,  0,  0,  0,  0,
};
```

It is now clear why having an array of three integers is a better representation than three different integers like in Yamashita's original implementation. By using the array `sq_to_bb_index` to access the bitboard integer that is needed, no if-statements are required. Furthermore, it is also clear why having the three integers representing the white promotion zone, the middle of the board and the black promotion zone is a better representation. In the current implementation, each rank on the board only has bits in one of the three bitboard integers, while in Yamashita's original implementation a rank on the board could have bits in more than one integer, needing extra operations to get the information for a single rank.

2.2.3 Vertical attacks

Calculating the attacks along a rook file (and lance file) is a little more complicated than the attacks along a rank. The problem is that for horizontal attacks, the bits are in consecutive positions, so masking and shifting can be used to extract these bits from the `Occupied` bitboard (this is done in the function `GetRankBits` above). For attacks along a file, the bits are 9 positions apart, so many more operations are required to extract the block pattern.

9	8	7	6	5	4	3	2	1	
0	1	2	3	4	5	6	7	8	a
9	10	11	12	13	14	15	16	17	b
18	19	20	21	22	23	24	25	26	c
27	28	29	30	31	32	33	34	35	d
36	37	38	39	40	41	42	43	44	e
45	46	47	48	49	50	51	52	53	f
54	55	56	57	58	59	60	61	62	g
63	64	65	66	67	68	69	70	71	h
72	73	74	75	76	77	78	79	80	i

9	8	7	6	5	4	3	2	1	
0	9	18	27	36	45	54	63	72	a
1	10	19	28	37	46	55	64	73	b
2	11	20	29	38	47	56	65	74	c
3	12	21	30	39	48	57	66	75	d
4	13	22	31	40	49	58	67	76	e
5	14	23	32	41	50	59	68	77	f
6	15	24	33	42	51	60	69	78	g
7	16	25	34	43	52	61	70	79	h
8	17	26	35	44	53	62	71	80	i

Figure 6: Normal square allocation on the left and 90 degree rotation square allocation on the right.

The solution to this problem is to use rotated bitboards. By rotating the board 90 degrees counter clockwise, the bits that were 9 positions apart, are now next to each other (see Figure 6)[1].

The 90 degree rotation to make the bits for vertically connected squares adjacent in the bitboard representation changes the attack calculation. Now the array `FileAttacks` is needed in which every rotated block pattern is translated to a vertical attack. The possible block patterns are the same as in the case of horizontal attacks, but instead of horizontal attacks, the array has to give the vertical attacks. An example for a rook on 2e is given in Figure 7. Square 2e is 43, the block pattern is $1001011 = 75$ (edge squares stripped off this time) and the bitboard for the attack for this block pattern has a 1 at bit 1 and bit 10 of the first integer, a 1 at bit 19 and 1 of the second integer and a 1 at bit 19 of the third integer:

```
FileAttacks[43][75].bb[0] = 2;
FileAttacks[43][75].bb[1] = 524292;
FileAttacks[43][75].bb[2] = 524288;
```

Furthermore, a new bitboard `Occupied_rot90` is needed to represent the occupied squares in case of rotation. The information in this bitboard must be updated at every move, giving a small overhead. The file attack code now is straightforward:

```
void CalcFileAtt(BITBOARD *bb, int from) {
   // Get the block pattern, rotating
   // the source square 90 degrees
   int pcs = GetRankBits(&Occupied_rot90,
                         init_l90[from]);
   // Strip the bits for the edge squares
   pcs = (pcs & CLEAR9MASK) >> 1;
   // Get the vertical attacks
   CopyBB(bb, &FileAttacks[from][pcs]);
}
```

[1] Yamashita uses a 90 degree clockwise rotation, but this is not a significant difference.

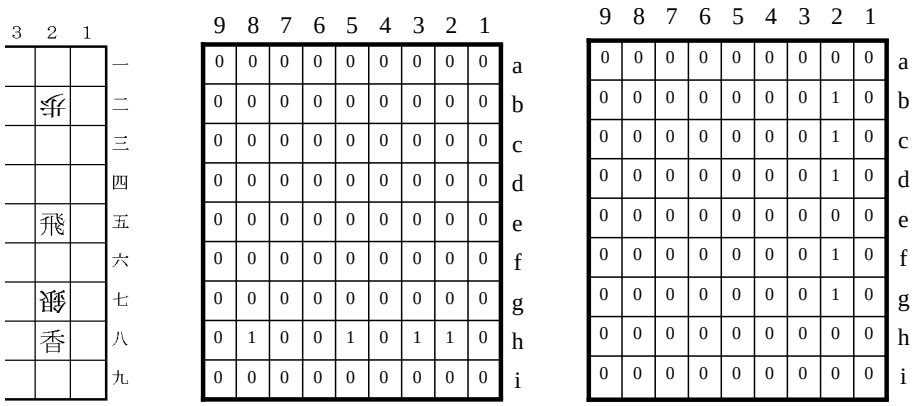

Figure 7: Example of a board position (left), its representation with a 90 degree anti-clockwise rotation (middle) and the bitboard representing the attacks of the rook on 2e (right).

```
// Translate square to 90 degree
// anti-clockwise rotation
int init_l90[] = {
   72, 63, 54, 45, 36, 27, 18,  9, 0,
   73, 64, 55, 46, 37, 28, 19, 10, 1,
   74, 65, 56, 47, 38, 29, 20, 11, 2,
   75, 66, 57, 48, 39, 30, 21, 12, 3,
   76, 67, 58, 49, 40, 31, 22, 13, 4,
   77, 68, 59, 50, 41, 32, 23, 14, 5,
   78, 69, 60, 51, 42, 33, 24, 15, 6,
   79, 70, 61, 52, 43, 34, 25, 16, 7,
   80, 71, 62, 53, 44, 35, 26, 17, 8,
};
```

This solution was given by Hyatt [2] for chess and it can be used in shogi as well. In chess it is only used for rook attacks, but in shogi it can be used for both rooks and lances.

2.2.4 Diagonal attacks

It is now clear how to find the attacking squares for rooks and lances, but there is still the problem of the bishop. Bishops attack along diagonals and the bits for diagonals are not adjacent. A rotation of 90 degrees will not help, but Hyatt pointed out that the same rotation trick can be used for bishops, but now rotations of 45 degrees clockwise and 45 degrees anti-clockwise must be used. In Figure 8, the rotations of the original squares (9a is 0, 1i is 80) are given.

In this figure it can be seen that for a 45 degree clockwise rotation, all diagonals going up from left to right are now adjacent (horizontal) squares. For example, the squares 27 19 11 3 are the diagonal that starts from 9d and ends at 6a. In contrast, for a 45 degree anti-clockwise rotation, the diagonals going down from left to right are adjacent. For example, the squares 4 14 24 34 44 are the diagonal that starts from 5a and ends at 1e.

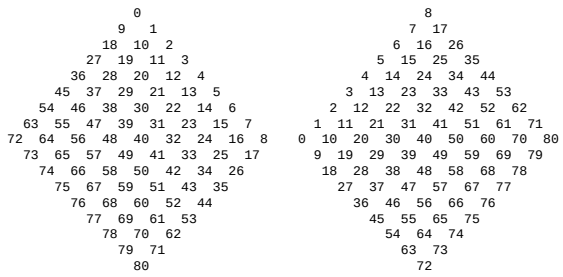

Figure 8: Rotating the original board squares 45 degrees clockwise rotation (left) and 45 degrees anti-clockwise rotation (right).

To put diagonals in adjacent bits of an attack bitboard, the bits of the bitboard are assigned as in Figure 9.

The procedure for calculating the squares attacked by a bishop is now similar to the procedure for rook and lance. For each possible block pattern of pieces on a diagonal, the attacks are pre-calculated. Furthermore, the bitboards `Occupied_r45` and `Occupied_l45` contain the occupied squares for a 45 degree clockwise rotation and 45 degrees anti-clockwise rotation respectively. These two bitboards must be updated for every move.

There is one extra problem in the case of diagonal attacks. In Figure 9 it can be seen that there are diagonals overflowing into the next integer of the bitboard. For clockwise rotation, square 6 is part of the diagonal 54 46 38 30 22 14 6 (9g to 3a) and square 74 is part of the diagonal 74 66 58 50 42 34 26 (7i to 1c). Similarly, for anti-clockwise rotation, the square 62 is part of the diagonal 2 12 22 32 42 52 62 (7a to 1g) and square 18 is part of the diagonal 18 28 38 48 58 68 78 (9c to 3i).

0	9	1	18	10	2	27	19	11
3	36	28	20	12	4	45	37	29
21	13	5	54	46	38	30	22	14
6	63	55	47	39	31	23	15	7
72	64	56	48	40	32	24	16	8
73	65	57	49	41	33	25	17	74
66	58	50	42	34	26	75	67	59
51	43	35	76	68	60	52	44	77
69	61	53	78	70	62	79	71	80

8	7	17	6	16	26	5	15	25
35	4	14	24	34	44	3	13	23
33	43	53	2	12	22	32	42	52
62	1	11	21	31	41	51	61	71
0	10	20	30	40	50	60	70	80
9	19	29	39	49	59	69	79	18
28	38	48	58	68	78	27	37	47
57	67	77	36	46	56	66	76	45
55	65	75	54	64	74	63	73	72

Figure 9: Bit assignment for 45 degrees clockwise rotation (left) and 45 degrees anti-clockwise rotation (right).

Square 74 (18 in the anti-clockwise version) is not a problem, because this is an edge square at the top of the diagonal and will be stripped anyway. Therefore, this overflow into the next integer of the bitboard (from `bb[1]` to `bb[2]`) can be ignored. Square 6 (62 in the anti-clockwise version) is also an edge square, so its contents can be ignored. However, because this time the square is at the bottom of the diagonal (i.e. the most significant bit) there is the problem of fitting the remaining 6 bits of the diagonal to the 7 bit block information. A single left shift is needed while for all the other diagonals a right shift is needed. Therefore, these two diagonals are a special case and a condition has to be added to the function that gets the block information from the rotated boards `Occupied_l45` and `Occupied_r45`.

The pre-calculated attack bitboards are stored in the following bitboard arrays:

```
BITBOARD DiaAtt_r45[81][128];
BITBOARD DiaAtt_l45[81][128];
```

The calculation procedure is now as follows:

```
void DiaAttRL(BITBOARD *bb, int from) {
  int pcs = GetDiaBits(&Occupied_r45,
                 init_r45[from]);
  pcs = (pcs & CLEAR9MASK) >> 1;
  CopyBB(bb, &DiaAtt_r45[from][pcs]);
}

void DiaAttLR(BITBOARD *bb, int from) {
  // Similar, with left 45 degree rotation
}

int GetDiaBits(BITBOARD *bb, int sq) {
  // Special case of 9g-3a and 7a-1g
  if(sq >= 21 && sq <= 27)
    return (bb->bb[0] & ALLONES_6) << 1;
  // Other cases
  return (bb->bb[sq_to_bb_dia[sq]]
     >> dia_shift_no[sq]) & dia_mask[sq];
}
```

```
int sq_to_bb_dia[] = {
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2,
  2, 2, 2, 2, 2, 2, 2, 2, 2,
};

int dia_shift_no[] = {
  26, 24, 24, 21, 21, 21, 17, 17, 17,
  17, 12, 12, 12, 12, 12,  6,  6,  6,
   6,  6,  6,  0,  0,  0,  0,  0,  0,
   0, 18, 18, 18, 18, 18, 18, 18, 18,
   9,  9,  9,  9,  9,  9,  9,  9,  9,
   1,  1,  1,  1,  1,  1,  1,  1, 21,
  21, 21, 21, 21, 21, 21, 15, 15, 15,
  15, 15, 15, 10, 10, 10, 10, 10,  6,
   6,  6,  6,  3,  3,  3,  1,  1,  0,
};

int dia_mask[] = {
    1,  3,  3,  7,  7,  7, 15, 15, 15,
   15, 31, 31, 31, 31, 31, 63, 63, 63,
   63, 63, 63,127,127,127,127,127,127,
  127,255,255,255,255,255,255,255,255,
  511,511,511,511,511,511,511,511,511,
  255,255,255,255,255,255,255,255,127,
  127,127,127,127,127,127, 63, 63, 63,
   63, 63, 63, 31, 31, 31, 31, 31, 15,
   15, 15, 15,  7,  7,  7,  3,  3,  1,
};
```

The `dia_mask` translation is needed because unlike ranks and files, diagonals do not have a fixed length. Therefore, masking is needed to keep only the information that has the precise length of the diagonal on which the particular square is located. The `init_r45` and `init_l45` arrays have been omitted. These arrays give the position of a square after 45 degree clockwise and anti-clockwise rotation.

2.3 Other Bitboards

The bitboards explained above are enough to generate almost all of the moves in shogi. For example, the moves for a black bishop can be found by first making a bitboard with 1s on the squares where the bishop from square `sq` can move to:

```
BITBOARD bb1, bb2, moves;
DiaAttRL(&bb1, sq);
DiaAttLR(&bb2, sq);
OrBB(&moves, &bb1, &bb2);
AndBB(&moves, &moves, &GotePieces);
```

Then, a loop through the bits gives the possible destination squares (for a promoted bishop, these

Version	*Time (s)*
Attack tables	24810
Bitboards	12709

Table 1: Results of 4 ply minimax searches in 100 positions.

destination squares are ADD-ed with a bitboard that has 1s for all king moves on `sq`). What remains is to decide if the piece can promote or not. For this, bitboards can be used that have 1s for the squares inside the black or white promotion zone. The promotion zone is represented by a single integer, so checking if a square is in the promotion zone can be done by checking only `bb[0]` or `bb[2]`. In the same way, bitboards can be used to check if a lance, knight or pawn must promote (lance and knight on the top two ranks and pawn on the top rank).

3 Experimental Results

In shogi programs, bitboards have not been as widely used as in chess, which is to be expected because of the difference in board size. It is more common to have attack tables (*kiki tables* in Japanese) that keep track of where each piece can move to, updating this information when the board position changes. Only after it became clear that the current Computer Shogi World Champion BONANZA is using bitboards, the discussion about its use in shogi is re-opened. Yamashita compared the use of bitboards with the use of attack tables in his program YSS and only found an increase in speed of about 20% [4] when bitboards are used in a tsume shogi solver.

Here a different test has been performed. To make a clean comparison between bitboards and attack tables, the performance of a minimax program in which the moves were generated by attack tables used in the program SPEAR was compared with a program in which the moves were generated by using bitboards. Therefore, the generated trees in both cases were exactly the same size. The evaluation function only calculated the material balance. In this way, a 4 ply minimax search was conducted for 100 different positions taken from two professional games. This experiment was run on a 3.0GHz Pentium 4 machine under WindowsXP. The results are summarized in Table 1.

From this table it can be seen that the move generation using bitboards is 48.8% faster than the move generation with attack tables.

4 Conclusions and Future Work

In this paper it was explained how bitboards can be used to do move generation in shogi. Even though the implementation of bitboards in shogi is less efficient than in chess because of the different board size (the 64 squares of a chess board fit in exactly one integer), it has been shown that the speed of move generation in shogi can be improved significantly if bitboards are used instead of attack tables. Using bitboards instead of the attack tables in SPEAR made the move generation almost twice as fast. Therefore, it seems likely that bitboards are a good alternative for attack tables.

This being said, move generation is not the only part of a shogi program where attack tables are used. For example, attack tables are also used in the evaluation function, especially when calculating piece mobility and the danger on squares around the king. It is possible that the extra calculations used in bitboards make the evaluation slower, thus decreasing the overall performance of the program. There is a big difference between the evaluation functions of different programs, so the effectiveness of the bitboard representation might have to be tested for every program.

However, there are other reasons for using bitboards. First, other parts of the program may also benefit from using bitboards. For example, the static exchange evaluator used in quiescence search can be implemented quite efficiently using bitboards. Second (and perhaps most appealing), if machines become available that allow 96 bit addressing, the shogi board will fit and the bitboard representation will become between two and three times faster overnight without much additional effort. Such an improvement cannot be expected in the case of attack tables.

References

[1] K. Hoki. http://524.teacup.com/yss/bbs. Message posted on August 20th 2005.

[2] R. Hyatt. http://www.cis.uab.edu/hyatt/bitmaps.html.

[3] Wikipidia. http://en.wikipedia.org/wiki/Bitboard.

[4] H. Yamashita. http://524.teacup.com/yss/bbs. Message posted on August 12th 2005.